資　料 4

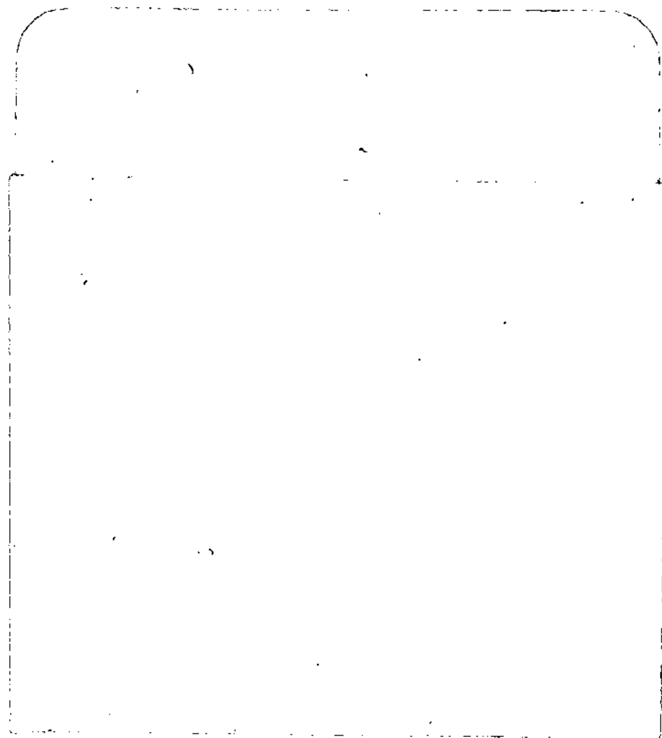
第５世代の電子計算機に関する調査研究報告書

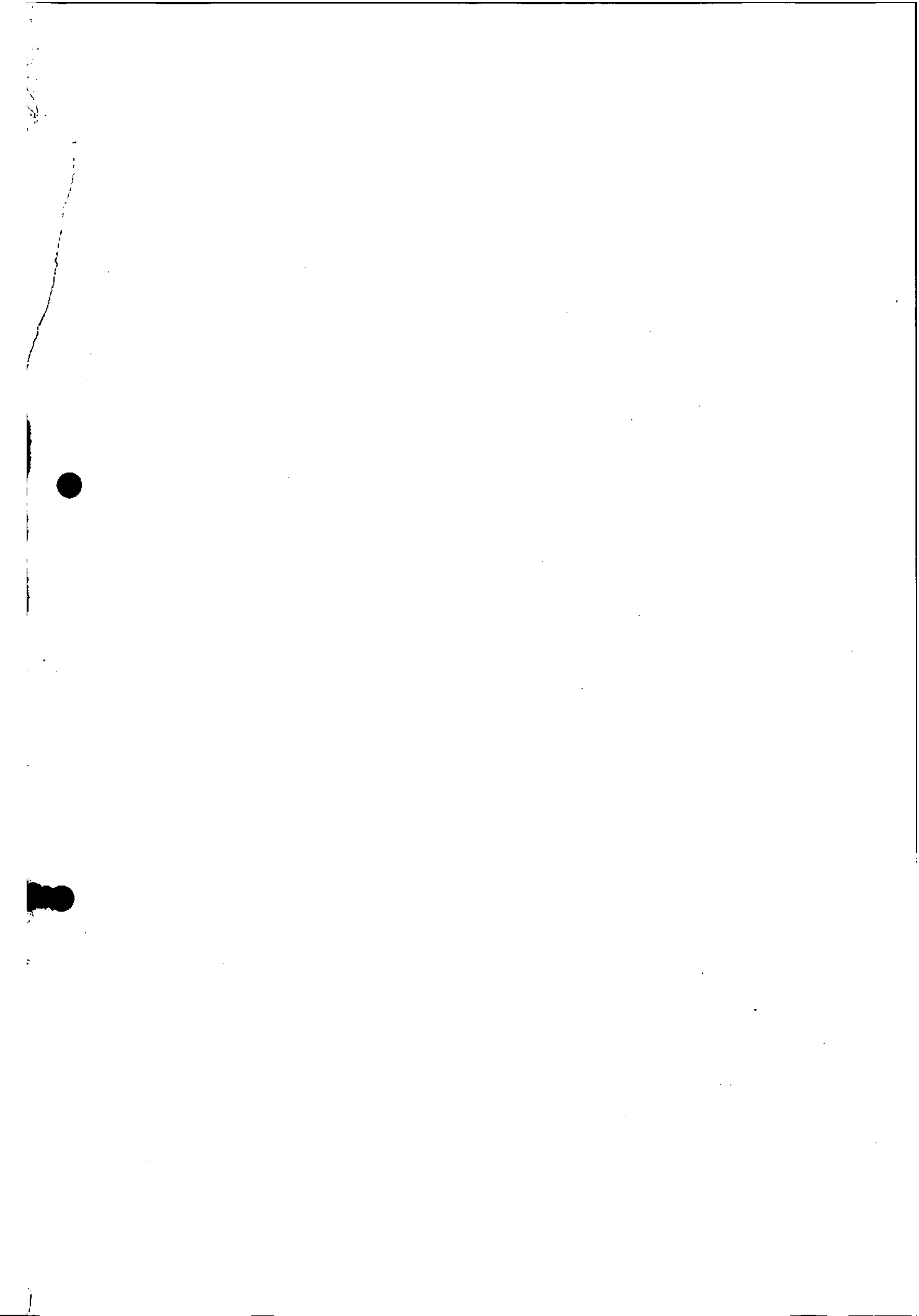
―マシンの理論―

昭和55年３月

JIPDEC

財団法人　日本情報処理開発協会





この報告書は，日本自転車振興会から競輪収益の一部である機械工業振興資金の補助を受けて昭和５４年度に実施した「第５世代の電子計算機に関する調査研究」の成果をとりまとめたものであります。

マシン理論執筆者

（順不同）

氏　　名	所　　　属
横　井　俊　夫	電子技術総合研究所　パターン情報部 推論機構研究室　主任研究官
佐　藤　泰　介	電子技術総合研究所　パターン情報部 推論機構研究室　研究員
元　吉　文　男	電子技術総合研究所　パターン情報部 推論機構研究室　研究員
大谷木　重　夫	電子技術総合研究所　制御部 論理システム研究室　研究員
山　口　喜　教	電子技術総合研究所　電子計算機部 計算機方式研究室　研究員
二　木　厚　吉	電子技術総合研究所　ソフトウェア部 言語処理研究室　研究員
内　田　俊　一	電子技術総合研究所　ソフトウェア部 情報システム研究室　研究員
塚　本　亮　治	電子技術総合研究所　制御部 情報制御研究室　研究員
樋　口　哲　野	慶応大学　工学部　電気工学科

目　　次

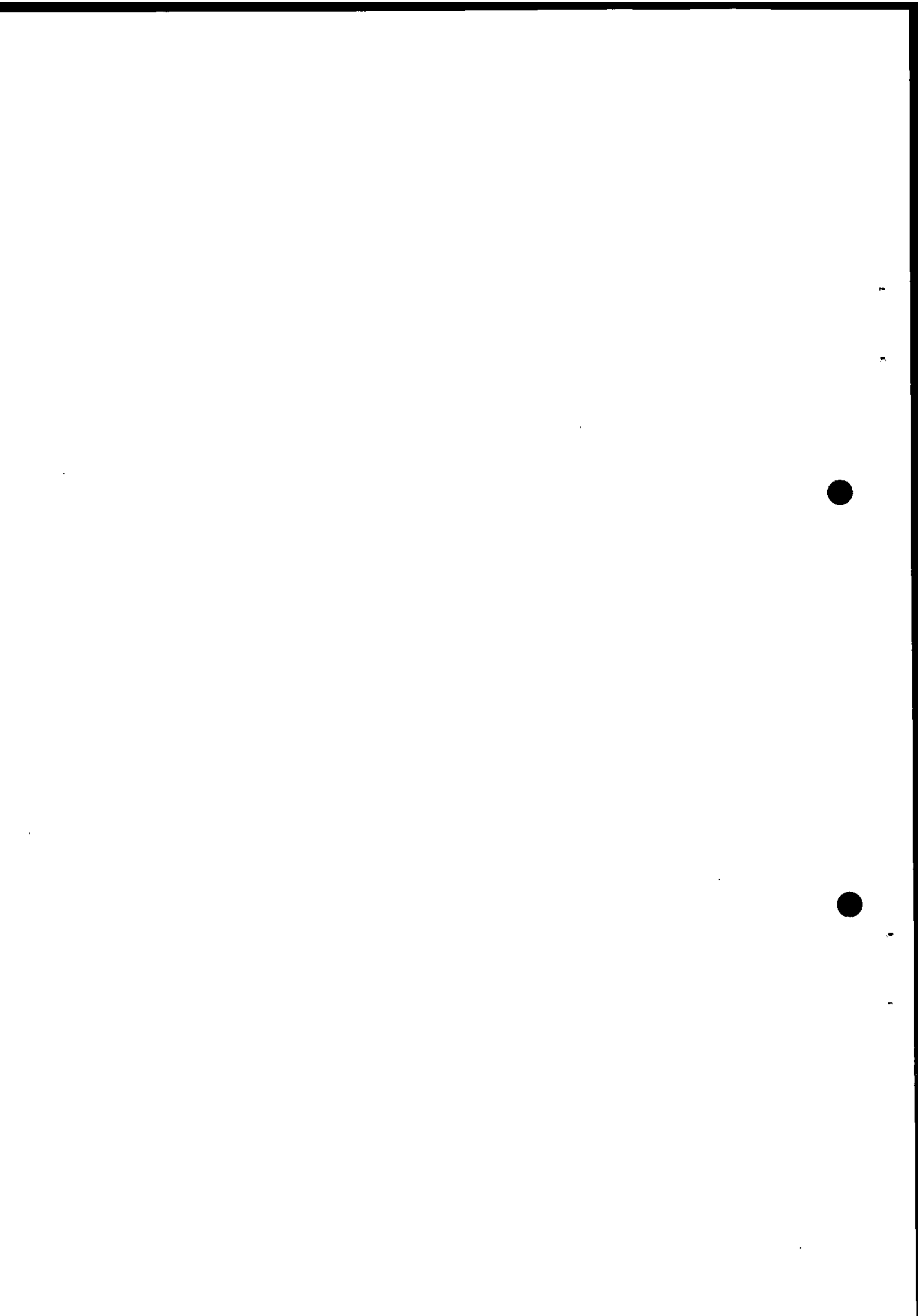
1. はじめに ………… 1
2. 鳥瞰図 ………… 3
3. 計算理論の諸相 ………… 5
3.1 はじめに ………… 5
3.2 Combinatory Logicの基本理念とその方法 ………… 6
3.3 計算システムとしてのλ−Calculusのもつ一般性 ………… 9
3.4 Computational LogicとしてのEquational Logic ………… 15
4. 論理プログラミングについて ………… 25
−背景とその周辺，実現法−
4.1 はじめに ………… 25
4.2 Clausal Logic ………… 29
4.2.1 Clausal Logicの導入 ………… 29
4.2.2 Clausal LogicとHerbrand Universe ………… 30
4.2.3 Clausal Logicの拡張 ………… 32
4.3 Horn Set ………… 33
4.3.1 Hornの一般的定義とmodel theory ………… 34
4.3.2 充足不能なHorn Setに対するRefutation Method ………… 36
4.4 論理プログラムのコンパイル ………… 41
4.4.1 正則Hornとコンパイル ………… 42
4.4.2 一般HornのΣへのコンパイル ………… 46
4.5 一般のclauseによる論理プログラミング ………… 48
4.6 おわりに ………… 49
参考文献 ………… 50
5. 代数的プログラミング ………… 53
−代数的仕様とその階層的プログラミングへの応用−

5.1 まえがき …… 53
5.2 代数的仕様とその意味 …… 55
5.2.1 なぜ代数か？ …… 55
5.2.2 代数的仕様 …… 56
5.2.3 書き換え規則に基づく意味定義 …… 59
5.3 代数的仕様に基づく
階層的仕様記述／プログラミング法 …… 61
6. 数式とプログラム …… 69
6.1 数式処理システム …… 69
6.2 代表的システム …… 71
6.2.1 MACSYMA …… 71
6.2.2 REDUCE …… 72
6.3 数式の変形 …… 72
6.4 プログラムと数式処理 …… 74
7. 関数型言語 …… 77
7.1 はじめに …… 77
7.2 関数型言語の一般的特徴 …… 78
7.3 λ計算 …… 80
7.3.1 λ計算の表現形式 …… 81
7.3.2 λ計算の規則 …… 81
7.3.3 Church-Rosserの定理と並列性 …… 82
7.4 関数型言語の評価構造 …… 84
7.4.1 再帰関数とスタックモデル …… 85
7.4.2 Funarg問題と木構造モデル …… 87
7.4.3 Lazy evaluation方式 …… 88
7.4.4 Continuation …… 89
7.5 関数型言語の形体 …… 91

7.6　むすび ………… 92
8.　駆動型プログラミング ………… 97
8.1　はじめに ………… 97
8.2　並列処理マシンを構築する動き ………… 99
8.2.1　まえがき ………… 99
8.2.2　データフローマシンの研究背景 ………… 100
8.2.3　並列処理の問題点 ………… 101
8.2.4　データフローモデルの性質 ………… 104
8.2.5　データフローマシンに対する課題 ………… 108
8.2.6　おわりに ………… 112
8.3　ソフトウェア工学との関連性 ………… 112
8.3.1　はじめに ………… 112
8.3.2　オブジェクトのモデル化 ………… 113
8.3.3　モジュール化 ………… 114
8.3.4　信頼性の高いコンポーネントの作成 ………… 115
8.3.5　言語構造やモデルの理論的研究 ………… 116
8.3.6　まとめ ………… 116
8.4　知識表現をめざす言語との関連性 ………… 117
8.5　おわりに ………… 119
9.　並行プログラミング ………… 123
(Concurrent Programming)
9.1　まえがき ………… 123
9.2　歴史的概観 ………… 124
9.2.1　高価なハードウェア ………… 124
9.2.2　高価なソフトウェア ………… 125
9.2.3　高価なシステム ………… 126
9.3　適用分野 ………… 127

9.3.1 リアルタイム・システム ………… 128
9.3.2 オペレイティング・システム ………… 128
9.3.3 分散システム ………… 129
9.4 プログラミングの概念 ………… 129
9.4.1 同期(手続)指向型 ………… 132
9.4.2 通信(メッセージ)指向型 ………… 138
9.4.3 仕様指向型 ………… 143
9.4.4 コルーチン指向型 ………… 146
9.5 言語の設計要件 ………… 152
9.5.1 モジュラリティ ………… 152
9.5.2 通 信 ………… 152
9.5.3 表現能力と使い易さ ………… 153
9.5.4 単一言語で十分か ………… 153
9.6 あとがき ………… 154
9.7 参考文献 ………… 155
付 録 代表的な並行プログラム言語 ………… 161~162
10. 対象指向型プログラミング ………… 163
10.1 Closure, Actor ………… 164
10.2 Class ………… 164
10.3 抽象データ型の抽象化 ………… 167
10.4 モニタと順路式 ………… 169
10.5 フレームと知識表現 ………… 169
11. 新計算機の骨組 ………… 173
12. 新プログラム言語の構造 ………… 177
13. むすび ………… 183

第1章　はじめに



1. は じ め に

いかなるプログラム言語に適したオブジェクト・コードを生成するのか，あるいは直接実行するのかによって，その計算機のアーキテクチャは決定される。IBM360，370，303X等，現在の汎用計算機の標準版であるアーキテクチャの流れも，FORTRAN，COBOL，PL／I という言語を対象とした流れと見ることができる。ALGOLを対象としたBurroughs の計算機，最近のLisp マシン，APLマシンにいたるにつれ，その傾向は顕著になりつつある。いわゆる高級言語マシン，その程度の差こそあれ，すべての汎用計算機は，汎用プログラム言語マシンとして設定されてきたと断言できる。

アーキテクチャが対象とする言語によって決定されるとしたら，その言語の骨組み，言語のアーキテクチャともいえるものは，何によって定められるのであろうか。それを定めるものが，マシンの理論ワーキング・グループが調査・研究の対象とした計算の機構と呼ぶものである。

新しい計算の機構を論じ，新しい計算機アーキテクチャを考案しようという試みは，決して新しいものではない。むしろ，いささか飽き飽きした議論でもある。チューリング・マシン，ポストのプロダクション・システム，λ一計算，帰納関数理論等々，計算機構の理論的支柱は，電子計算機の出現以前に列挙された。そして，チューリング・マシンに基づいた計算機アーキテクチャの確立と後にIBM型計算機として不動の地位を獲得していく流れの初めから，別の計算機構に基づくよりすぐれたアーキテクチャをという試みが行なわれた。しかし，いずれも著しい成果を得ずに終る。Burroughs の孤独な抵抗であるALGOLマシンも，売るためには，その上にFORTRANコンパイラを乗せねばならなかった。

それでは，何故計算機構を，あえて今，議論するのであろうか。第五世代計算機のアーキテクチャを，あえて，その大本の計算機構から論じようとするのは，今までとは違った新しい状況が生じつつある，今，情報処理技術の流れに

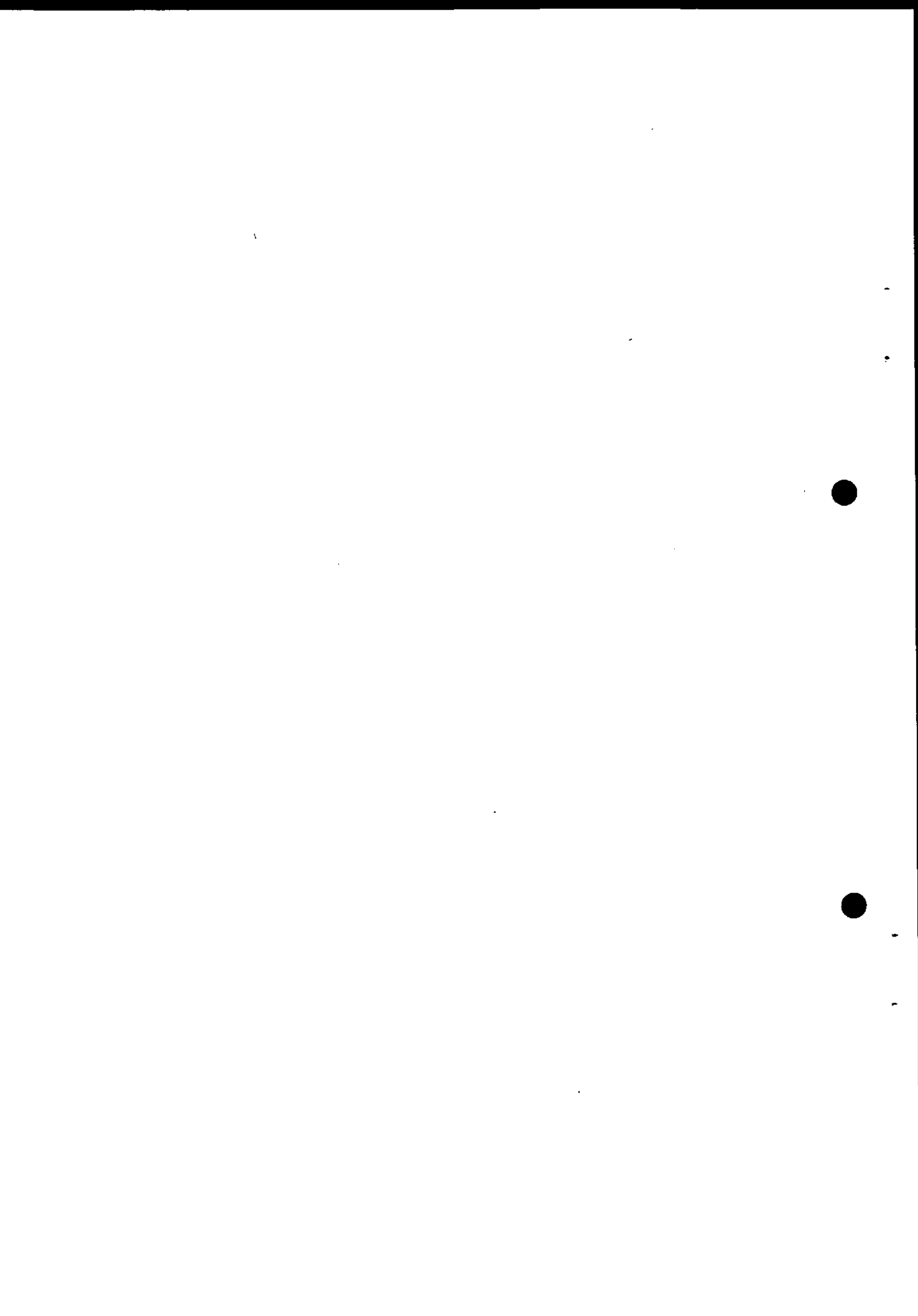
新しい変曲点が生じつつあるという確信からである。

まず第一に，新しい計算機構，新しいプログラム言語への動きである。1970年頃から再び復興した人工知能研究は，着実な努力が続けられ，高度な利用形態を可能にしつつある。その過程で，人間の持つ各種の知識を表現しようという研究は，新しい記述系を提案し，新しい計算の機構を確立するまでになった。同じく，１９７０年頃から，プログラムの意味を厳密に定め，正しさの証明を理論的に行おうという研究が本格化した。まず，既存のプログラム（言語）に適用された。結果は，それ程芳しいものではなかった。理論の未熟さを克服する努力とともに，既存言語に対する強い反省が起った。さらに進んで，新しい計算機構，言語を提案するまでになった。

第二は，素子技術の進歩である。その進歩は，極端な低廉化と均一化である。主記憶，処理装置が，それぞれ別々の素子技術で作られ，どれもが非常に高価であった時代は，この事実を陽に認めた計算機構が優位を占める。チューリング・マシンの理論とFORTRAN，COBOLである。VLSI技術の進歩は，この事実を急激な勢いで遠い過去のものとしつつある。さらに均一化は，処理装置と主記憶装置を，数の上でも機能的にも同質化し，従来のハードウェア技術では考えられなかったような，高度な並列処理アーキテクチャを可能にしつつある。

新しい計算機構，新しい計算の記述系への提案がなされてきたということと，その提案を素直にアーキテクチャ上に実現しうるような素子技術が成熟しつつあるという２点が，マシンの理論グループの調査・研究の指針である。

第２章　鳥瞰図



2. 鳥瞰図

代表的な計算機構を列挙し，それぞれの簡単な特徴，相互の関係を概観する。

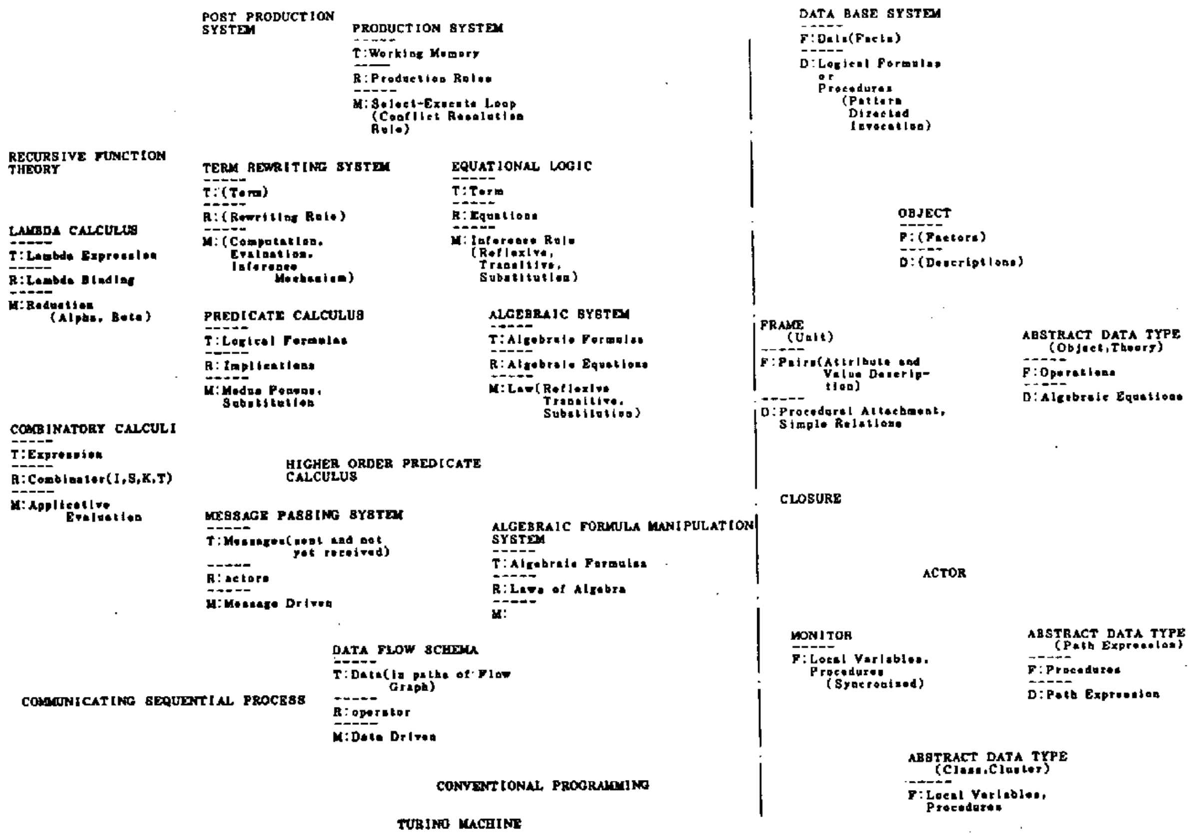


図2－1　計算の機構

図２－１に代表的な計算機構（理論）を列挙する。一点鎖線の左側は，action oriented な見方で見た場合，あるいは見方に入る計算機構である。右側は，object oriented な見方で見た場合，あるいは見方に入る計算機構である。

action oriented な見方に対しては，最も無構造で一般的と思われる書き換えシステム（Term Rewriting System）に対応させるとどうなるかということで，その特徴が付記されている。何がタームに対応し(T)，何が書き換え規則に対応し(R)，規則をタームに適用する機構はどうなっているか(M)が略記されている。

object orientedな見方に対しては，そのobject を記述する基本的な要素(F)が何で，その要素間の記述(D)がどう与えられるかによって述べられている。

図２－２は，簡単に関連を表示してある。矢印の向きは，おおむね，簡単なものから複雑なものへという方向づけになっている。数字は，各計算機構が詳述されている章番号を表わす。

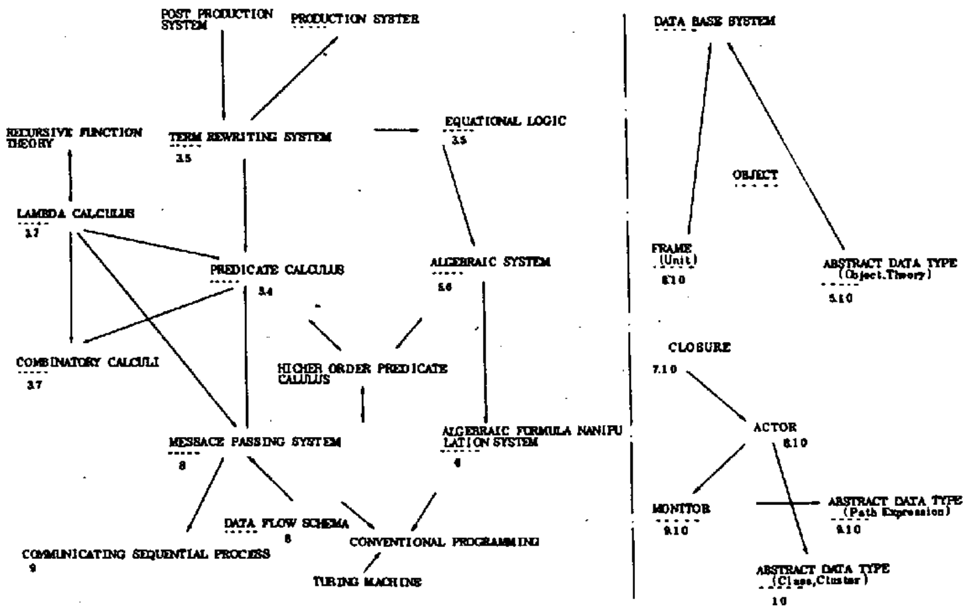
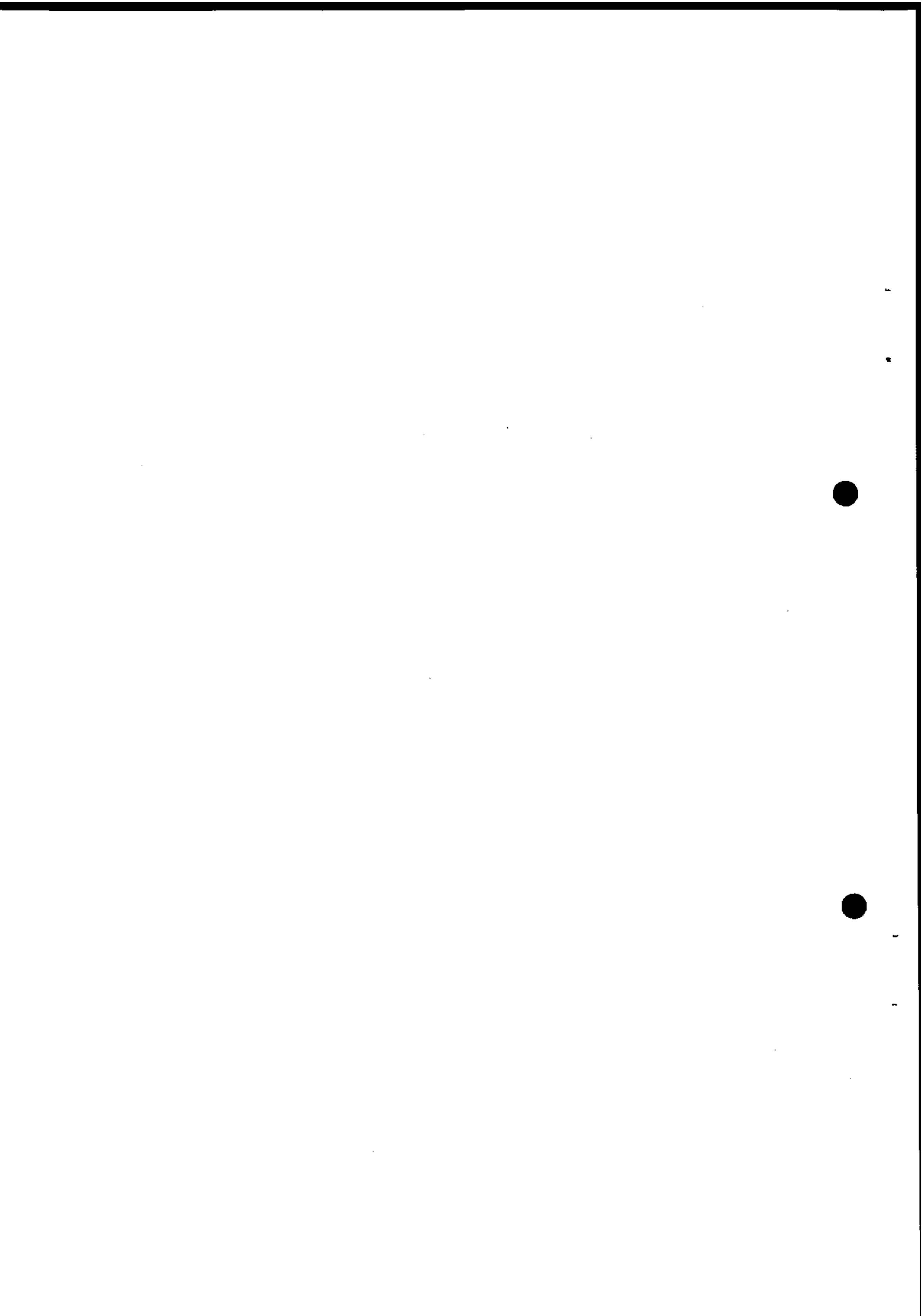


図２－２　相　関　図

第3章　計算理論の諸相



3. 計算理論の諸相

3.1 はじめに

1980年初頭に当って今後の情報処理技術の発展を考えるとき，超LSI技術がその大きな柱となることは誰もが疑いを入れないところであろう。問題はDevice側の急速な進歩を吸収するシステム技術のあり方である。

Hardware技術としては

① Multi-processor system

② Data Flow processor system

Software技術としては

① Concurrent programming

② Production system

等が今のところ主な候補としてあげられるであろう。

最も現実的な立場に立てば

multi-processor system

+

concurrent programming language

という図式に従う選択が賢明であろう。

又，より革新をめざすならば

Data-Flow processor system

+

Production系言語

という組合わせが考えられる。

またuser側に見える言語という意味では

① Lisp, λ－Calculus

② Abstract Data type, Equational Logic

③ Logic programming

等の並列処理系を考えるのは魅力的な研究課題である。

本稿では

① Production系言語

② λ－Calculus

③ Equational Logic

に共通する計算メカニズムとして書換規則をとりあげ，書換システムとしてλ－CalculusやEquational logicを説明することによって，それらがproduction 系言語の上にfitする事を示したい。そのためにはλ－CalculusをCombinatory logicとしてとらえるのが歴史的にも，また実際的にもよい。更にその発展系としてequational logicをとらえることもできるというわけである。

3.2 Combinatory logicの基本理念とその方法

Schönfinkelが最初にcombinatory logicを提唱したときの意図は，一階の論理式を表現し得る最小の基本関数系を求めることにあったようである。意図は極めてclassicであるがその定式化の過程で得られた概念は極めて現代的な意義をもつことになった。

① 自由変数を完全にシステムから消去し，そのかわりにsubstitutionを演算の基本とする。

② 全ての論理式（ここでは自由変数を全称化により消去してあるのでclosed formulaのみ）を少数の基本関数の合成として表現する。

③ ②を遂行するため全ての論理式及びlogical connectiveを関数を関数へ移す一変数の関数として取扱う。

④ 基本関数に対しては書き換え規則を与える。

命題論理ではsheffer's stroke｜が全ての論理関数を生成する。そこで

$$\mho\ PQ = \forall x\ \ P(x) \mid Q(x)$$

とおくと一変数の closed formula を全てUと primitive predicate で表わせる。しかし多変数になると話は別である。

$$\forall x \quad \exists y \quad P(x, y) \wedge Q(y)$$
$$\equiv \forall x \neg \forall y \neg [P(x, y) \wedge Q(y)]$$
$$\equiv \forall \quad \neg \forall y \quad P(x, y) \mid Q(y)$$
$$\equiv \forall x \neg U y (P(x, y), Q(y))$$
$$\equiv \forall x U y (P(x, y), Q(y)) \mid U y (P(x, y), Q(y))$$
$$\equiv U x (U y (P(x, y), Q(y), U y (P(x, y), Q(y)))$$

と変形できるがここで式は closed であるから変数x，yは本来消去できるはずであるにもかかわらずこの式でx，yを消去した

$$U(UPQ)(UPQ)$$

なる式は意味をなさない。

③に従って

$$P(x, y) = P(x)(y)$$

即ちPという関数は

$$x \mapsto P(x)$$

P(x)という関数は

$$y \mapsto P(x)(y) = P(x, y)$$

なる一変数の関数と見なすこととする。

すると

$$\forall x \exists y \quad P(x, y) \wedge Q(y)$$
$$= \forall x \exists y (Px) y \wedge Q y$$
$$= \forall x \neg \forall y \neg [(Px) y \wedge Q y]$$
$$= \forall x \neg U (Px)(Q)$$

ここで式が$\forall x \quad \varphi x$ の形になってくれれば

$$\forall x \neg \quad \varphi x \quad = \quad \forall x \varphi x \mid \varphi x$$

$$=\mho\varphi\varphi$$

となって望み通りの形が得られる。

そのために関数適用の順序を変更する関数を導入する必要がある。

(i) $Ix = x$

(ii) $(Zf)gx = f(gx)$

(iii) $Kxy = x$

(iv) $(T\varphi)xy = \varphi yx$

(v) $S\varphi\chi x = (\varphi x)(\chi x)$

なる関数を導入すると

$$\mho(Px)(Q)$$

$$= (T\mho)(Q)(Px)$$

$$= (Z(T\mho)(Q))(P)(x)$$

であるから

$$= (Z((T\mho)(Q)))(P)$$

とおけばよい。

簡単な計算により

$$I = SKK,\ Z = S(KS)K,\quad T = S(ZZS)(KK)$$

が示されるから，全ての closed な論理式は $\mho$，K，S と constant predicate で表わされたことになる。

Schönfinkel に於いては　に対して書き換え規則が与えられていない点が combinatory logic としては不備である。

Curry は Schönfinkel の思想を押し進めて，Combinatory logic の次のような定式化を行なっている。

① system は有限個の atom をもつ。

② system は有限個の axiom をもつ。

③ system の object は atom か object を object に apply したものである。

④　system の sentence は X を object として

$\vdash X$

という形にかぎられる。

⑤　書き換え規則は

$\vdash A \rightarrow \vdash B$

乃至　$\vdash A_1 , \vdash A_2 \rightarrow \vdash B$

という形にかぎられしかも書き換えは一意的である。

K，S を基本 primitive としてもつ combinatory logic を Curry に従って定式化すると

I_1　$IX \vdash X$

I_2　$X \vdash IX$

K_1　$Z(KXY) \vdash ZX$

K_2　$ZX, EY \vdash Z(KXY)$

S_1　$V(SXYZ) \vdash V(XZ(YZ))$

S_2　$V(XZ(YZ)) \vdash V(SXYZ)$

となる。

3.3　計算システムとしての λ－Calculus のもつ一般性

A. Church の発見は K－S，combinatory logic に於ける証明が計算 system としても十分であるということにあるという見方も可能であろう。Schönfinkel の原理③に従って，対象を全て一変数の関数空間にとるものとする。また関数は全て closed な形式とするためその変数を論理式ではないので V のかわりに λ でしばるものとする。

$\lambda a \lambda b \ a(b)$

といったものがそういう意味での関数である。現われる関数は全て変数でしかもそれらは全てしばられているものとする。といわゆる λ－Calculus の式が

得られる。

ここで例えば

$$a(b) = ((Ka)b)(Ib)$$
$$= S(Ka)Ib$$
$$\lambda b\, a(b) = (TS)I(Ka)$$
$$= Z((TS)I)Ka$$
$$\lambda a \lambda b\, a(b) = Z((TS)I)K$$

であるが，このようにλ－Calculusの閉式は全てK，S－combinator で表現できるのである。そして演算は代入による書換規則の適用ということになる。

λ－Calculusの中で計算体系をつくるには，まず数体系を定義しなければならない。

$$0 = \lambda x \lambda y\, y$$
$$1 = \lambda x \lambda y\, x(y)$$
$$2 = \lambda x \lambda y\, x(x(y))$$

この数体系のもとで部分帰納関数を生成する基本関数を作ればよいわけである。

例えば

$$plus = \lambda x \lambda y \lambda a \lambda b\ \ (x(a))((y(a))(b))$$
$$times = \lambda x \lambda y\ \ x(y)$$

等とすればよい。

$$(plus\ 1)\ (2)$$
$$= \lambda a \lambda b((\lambda x \lambda y x(y))(a))(((\lambda x \lambda y x(x(y)))$$
$$(a))(b))$$
$$= \lambda a \lambda b(\lambda y\ \ a(y))(\lambda y a(a(y)))(b)$$
$$= \lambda a \lambda b(\lambda y a(y))(a(a(b))$$
$$= \lambda a \lambda b\ \ a(a(a(b))))$$
$$= 3$$

primitive recursionは

$$Y = \lambda u \lambda x u(x(x))(\lambda x u(x(x)))$$

で与えられる。

pure λ−calculus をそのまま計算体系にもってきたのではもちろん芸がなさすぎる。

① 数体系の表現が compact でない

② symbol identification の機能にかける

しかしこれらの点は容易に解決のつく問題である。

$$(\text{cons } a)(b) = \lambda z(z(a))(b)$$

$$\text{car } x = x(\lambda a \lambda b a)$$

$$\text{cdr } x = x(\lambda a \lambda b b)$$

によりλ−Calculus の中に List を define できるから

$$0 \longrightarrow (0)$$

$$1 \longrightarrow (1)$$

$$2 \longrightarrow (10)$$

$$\vdots$$

とする。

$$\text{cond}(x, y, z) = x \times Ky + (1 - x) \times Kz$$

とおくと

$$\text{cond}(1, y, z) = y$$

$$\text{cond}(0, y, z) = z$$

となるから，この cond 文でビットリストから数の識別をして計算ができる。

λ−Calculus の機能はこのような計算機能としての万能性，recursion, symbolic computation の能力の他に

① data type の識別機能

② program construct の構成機能

という2つの有効な能力がある。

例えば

$$\text{pair} = \lambda u \lambda z(z(\text{car } u))(\text{cdr } u)$$

とおく

$$\begin{aligned}
&\text{pair }(\text{pair})\\
&= \lambda z(z(\text{car pair}))(\text{cdr pair})\\
&= \lambda z(z[\lambda x \lambda y x\{\lambda u \lambda z(z(\text{car } u))(\text{cdr } u)\}]\\
&\quad \{\lambda x \lambda y\ \ y(\lambda u \lambda z(z(\text{car } u)(\text{cdr } u))\})\\
&= \lambda z(z\{\lambda u \lambda z(z(\text{car } u))(\text{cdr } u)\})\\
&= \lambda u \lambda z(z(\text{car } u)(\text{cdr } u))\\
&= \text{pair}
\end{aligned}$$

であり，pairはScottのいうretractでdata typeを規定する。

実際

$$\text{pair }((\text{cons } u)v) = (\text{cons } u)v$$

でpairはλ式zが（u．v）の形になっているかどうかを識別する。

また第2の点に関しては

$$n = \lambda x \lambda y\ \overbrace{x(\cdots\cdots x}^{n}(y))$$

に於いて

$$n\ (a)(b)$$

を行なえばこれはデータbに対しaをn回行なうDo-loopである。

λ−Calculus のもつ豊かな能力については以上で一応良いであろう。そこでここでは，λ−Calculus のもつ現代的意義について考察しよう。

計算機科学に於いて近年問題となったことがいくつかある。そのうちのいわゆるSoftware Crisis に関連しては

① プログラムをわかりやすく書く

1.1 Structured Programming

1.2 Control Flowの明確化（Go To文の除去）

② 虫とりの自動化

2.1　Automatic Programming

2.2　Automatic 検証系

2.3　単一割当＝side effect free 言語

並列処理に関連して

3.1　並列性制御言語＝Concurrent Programming

3.2　並列実行型言語＝Production system, Data Tlow

3.3　Dead Lock 検出

等の研究が進められてきた。

こういう研究の観点からλ−Calculus を見るとき

① Dead Lock-free 並列実行可

② 単一割当言語

③ explicit な制御言語をもたない

④ 複雑な問題を比較的簡潔に記述できる

等のすぐれた機能を備えていると見ることができる。

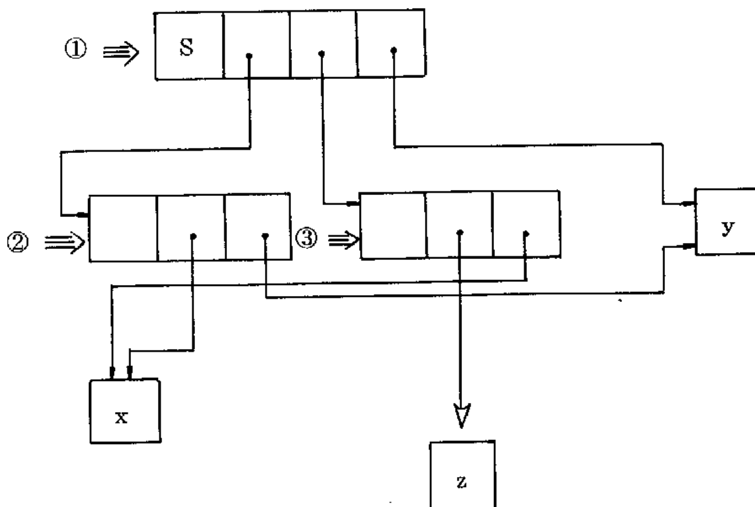
簡単のため

S(Kxy)(Kzx)(y)

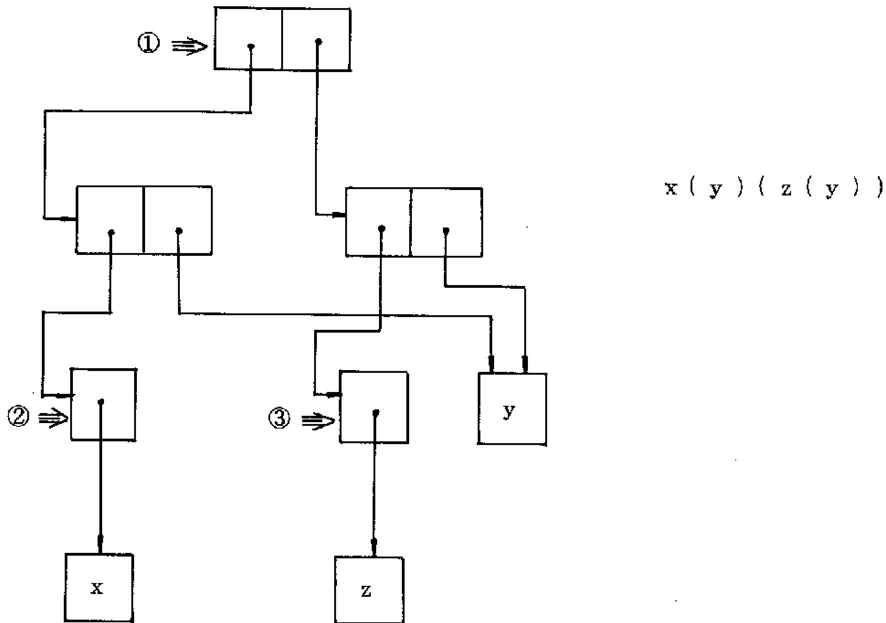
なる例を考える。(のちに詳述するが)いわゆるChurch Rosser property

により，この式はどこから評価してもよいし，一度にやってもよい。

内部表現を



とすると並列処理の結果は



となるがその際実行可能な①，②，③の３つの関数はそれぞれの実行が他の実行に影響を与えないのである。②，③については明白である。④についてはＬＣＦの試みを紹介すれば十分であろう。

Milner は conventional なプログラミング言語（例えば Pascal ）の検証系を次のようにして設定した。

① Pascal のプログラム construct に λ− Calculus で semantics を与える。

② 問題の spec を λ−Calculus で書く

③ 問題を解決する Pascal プログラムを書く

④ Pascal プログラムをその semantics に従って λ−Calcslusに落とす

⑤ ２つの λ−Calculus の式の等価性をチェック

この試みは②の問題の spec を λ−Calculus で書くことが比較的簡単に行なわれなければあまり意味のないもので λ−Calculus の記述力の高さを利用しているのである。

3.4 Computational logic としての equational logic

K－S combinatory logic を equational logic として書いてみよう

① WFF（整式）の定義

1.1 K，S はWFF

1.2 X，Y がWFF ならX（Y）もWFF

② 公 理

2.1 $x = x$

2.2 $Kxy = x$

2.3 $Sxyz = x(z)(y(z))$

③ Deduction

3.1 $\vdash x = y \Rightarrow \vdash y = x$

3.2 $\vdash x = y$ かつ $\vdash y = z \Rightarrow \vdash x = z$

3.3 $\vdash x = y \Rightarrow \vdash z(x) = z(y)$

3.4 $\vdash x = y \Rightarrow \vdash x(z) = y(z)$

ここでDeduction の性質から公理の等式の左辺と右辺のうちいづれかを標準形とみなして（ここでは右辺）

例えば

(i) $Kxy \longrightarrow x$

(ii) $Sxyz \longrightarrow x(z)(y(z))$

なる書き換え規則をもうけることとする。与えられた整式に対してこの書き換え規則を適用していくと最終的には一つの（とまらない場合もあるが）標準式になるであろう。このlogic の決定問題は与えられた。

$$X = Y$$

という等式に於いてX，Y それぞれを書き換え規則で書き換えていくことにより得られた標準形 X^*，Y^* が一致しているかどうかを調べることによって解ける。（解けない場合というのは書き換えがとまらなくて，X^* か Y^* のどちらかが得られないとき）

λ−Calculus では書き換え変形自身を計算とみ，標準形を計算結果とみているわけである。

さて，ここで述べたような枠組にあてはまるものは λ−Calculus にとどまらない。最近，プログラムの仕様記述及び検証用の言語として，Abstract Data Type に基づくものが有効な手段として認められてきている。

Abstract Data Typeによる仕様記述の目的とその適切性についての議論は別としてその概観を述べると

① 問題の解を数学的に厳密な公理系として設定し，これをプログラムの目標となる仕様として採用する。

② 公理系はData type とその間の関数に対して関数方程式として提示する。

③ 仕様の妥当性は公理系がequational logic で書かれているのでsample 計算をしてみることで調べる。

④ Program の妥当性は仕様にある公理をみたすかどうかでCheck する。

例えばAbstract Data Typeの考えに従ってstack というData Typeを定義すると

① Data Type

1.1 stack

1.2 item

1.3 boolean

② Function

2.1 newstack : Λ → stack

2.2 undefined : Λ → item

2.3 push : stack × item → stack

2.4 top : stack → item

2.5 pop : stack → stack

2.6 isnew : stack → boolean

③ Axiom

3.1 isnewstack (newstack) = true

3.2 isnewstack (push (s, i)) = false

3.3 pop (newstack) = newstack

3.4 pop (push (s, i)) = s

3.5 top (newstack) = undefineel

3.6 top (push (s, i)) = i

Abstract Data Typeによるこのような仕様記述は一つのequational logic を提示したものとなっており，従ってそれ自身計算systemとしての可能性を有している。

このようにequational logicはK－S combinatory logic にとどまらない広い範囲をcover するものである。しかし一般のequational logicを書き換えsystemにするとWFF X を書き換えた結果がただ一つの標準形X^*に至るとはかぎらない。即ち書き換えの順序によっては別の標準形X_1^*になってしまう場合がある。

このような点から書き換え系を一般的に考察する必要があるわけで，それをO′Donnellに従って述べよう。O′Donnellはsubtree replacement systemとして書き換えsystemを定式化した。

① Σを任意のsetとする

② Aは｛$\alpha \rightarrow \beta$｜ α，βはΣ－tree｝の有限subsetで書き換えの一意性即ち

$$\alpha \rightarrow \beta,\ \alpha \rightarrow \gamma \in A \Rightarrow \beta = \gamma$$

なる条件を満たす。

③ Deduction Rule

T_1 をΣ－tree，rをそのselector，T_1/rをrでselect されたT_1のsubtreeとする。

$T_1/r \rightarrow \beta \in A$ならば$T_1$のr－subtreeを$\beta$で置き換えたものを$T_2$

とするとき

$$T_1 \Rightarrow T_2$$

である。

Equational Logicから常にsubtree replacement systemを導くことができるという点には問題ないと思うが念のため。

Equational Logicの公理に

$$A = B,\ B = C,\ A = D$$

等があった場合には例えばAを標準にとり

$$B \rightarrow A,\ C \rightarrow A,\ D \rightarrow A$$

なる書換規則をもうけるものとする。(書換の一意性を保障)

Equational Logicの公理は関数方程式で与えられるから例えば

$$f(x_1, x_2, x_3)$$

を

$$((((f)(x_1))(x_2))(x_3))$$

なるbinary treeと見なすか

$$((f)(x_1 . x_2\ x_3))$$

なるmulti-treeと見なすかはいづれにせよとにかくtreeと見なせることもよかろう。

さてO' Donnellは任意のSRSシステムに於いてそれが高々一つしか標準形をもたないことをConfluence Propertyと呼んだ。

即ちSRS systemがconfluence propertyをもつとは

$$A \overset{*}{\Rightarrow} B,\quad A \overset{*}{\Rightarrow} C$$

ならば或るDがあって

$$B \overset{*}{\Rightarrow} D,\quad C \overset{*}{\Rightarrow} D$$

とできることである。

λ-Calculusに於いてはこれはChurch-Rosser propertyと呼ばれる。

Confluence propertyはこのままではテストできないから有限の範囲で

テスト可能な十分条件を得る必要がある。これが次に述べる可換性条件である。

SRS system が可換であるとは次の条件が成立することである。

$$A \rightarrow B \in \mathcal{A} \qquad A/x \rightarrow \beta \in \mathcal{A}$$

のとき

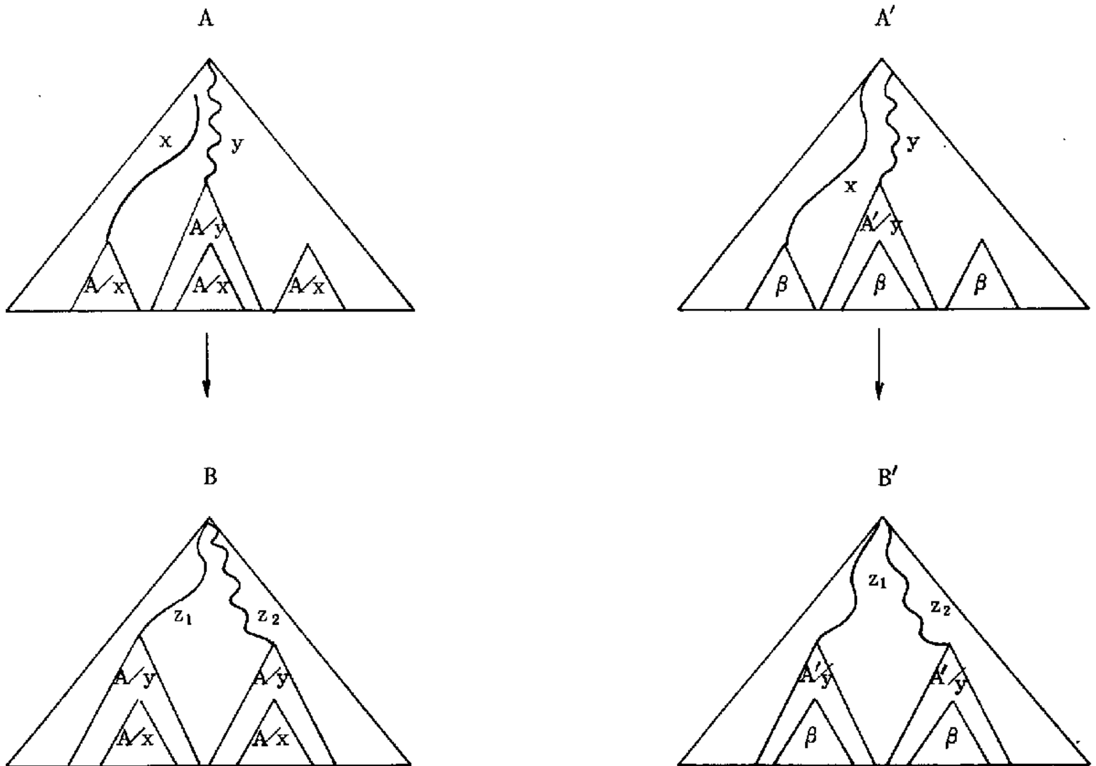
AのA/x−subtree を全てβで入れ換えたものをA′，BのA/x−subtreeを全てβで入れ換えたものをB′とするとき

$$A' \rightarrow B \in \mathcal{A}$$

が成立し

$$\exists \gamma \quad A/y \rightarrow \gamma \in \mathcal{A}$$

なる任意の selector y についてBのA/y subtree の位置とB′のA′/y subtree の位置が全体として完全に一致する。図で示すと



となる。

Aを関数tree と仮定するAのA/x subtree に自由変数aを代入してλでしばった関数をλaFとする。

すると

$$A = (\lambda a F)(A/x)$$

である。AのA/x−subtreeのselectorはaが現われる場所を示す。

いまA→Bの書き換えがλaF部分の書き換えになっていれば

$$B = (\lambda a G)(A/x)$$

$$A' = (\lambda a F)(\beta)$$

$$B' = (\lambda a G)(\beta)$$

であるから

同一の書き換えでA'→B'が得られねばならないから

$$A' \rightarrow B' \in \mathrm{Axiom}$$

が要求されるわけである。

これは大まかに言えば関数のbody側の書き換えとargument側の書き換えが独立に行ないうるといっているものと解釈できよう。

このような条件のもとでconfluence property が成立することを示すのは，そうむづかしいことではないので省略。

<参考文献>

〔1〕M. J. O'Donnell : "Computing in Sytems Described by Equations" Lecture Notes in Computer Science 58 1977

〔2〕B. K. Rosen : "Tree-Manipulating Sytems and Church-Rosser Theorems" JACM 20:1(1970)160~187

〔3〕G. C. Wraith : "Lectures on Elementary Topoi" Lecture Notes in Mathematics 445 p115~p206 1973

〔4〕M. Fourman : "The Logic of Topoi" in Handbook of Mathemalical Logic ed K. J. Barwise (Springer) 1977

〔5〕J. A. Goguen, J. W. Thatcher, E. G. Wagner : "Initial Algebra Semantics and Continuons Algebras" IBM Research Report RC 5701

〔6〕M. Schönfinkel : "Über die Bausteine der Mathematischen Logic" Math. Ann. 92 p305~p316 1924

〔7〕H. Curry : "Combinatory Logic" : 1974 vol1 (North-Holland)

〔8〕A. Church : "The Calculi of Lambda-Conversion" Annnals of Mathematics Studies 6 1941

〔9〕M. Davis : "Computability and Unsolvability" 1958 (McGraw-Hill)

〔10〕 D. Scott : "Assigning Probabilities to Logical Formulas" in Aspects of Inductive Logic 1965 (North Holland)

〔11〕 D. Scott : "Boolean-Valued Models for Set Theory" Lecture Notes for the Amer. Math. Soc. 1967

〔12〕 : "Advice on Modal Logic" in Philosophical Problems in Logic 1970 (Dordrecht)

〔13〕 : "Outline of a Mathematical Theory of Computation" Proc. 4th Ann. Princeton Conf. on Information Sciences and Systems p169~176 1970

〔14〕 : "The lattice of flow diagrams" Synposium on Semantics of Algorithmic Languages, Lecture Notes in Math. vol. 188 (Springer) 1971.

〔15〕 : "Continuons lattices" Lecture Notes in Math vol. 274 (Springer) 1/72

〔16〕 : "Lambda Calculus and Recursion Theory" Proc. 3rd Scandinavian Logic Symposium p154—193 (Narth Holland)

〔17〕 : "Combinators and classes" in λ-Calculus in Computer Seience Theory vol. 37 (Springer)

〔18〕 J. E. Donahue : "Complementary Definitions of Programming Languages Semantics" Lecture Notes in

Computer Science vol. 42

〔19〕 J. E. Stoy : "Denotational Somantics" 1977 (MITPRESS)

〔20〕 R. Milner : "LCF : A Way of Doing Proofs with a Machine" Lecture Notes in Computer Science vol. 74 p146~159

〔21〕 L. Aiello, M. Aiello, R. W. Weyhrauch : "Pascal in LCF Semantics and Example of Proof" Theoretieal Computer Science(1977) vol. 5 p135~p177

〔22〕 G. D. Plotkin : "LCF Cosidered as a Programming Language" Theoretical Computer Science vol. 5 (1977) p223~p255)

〔23〕 S. A. Ward : "Functional Domains of Applicative Languages" MACTR−136 1974

〔24〕 R. D. Jenks : "MODLISP : An Introduction" Lecture Notes in Comp. Sci. vol. 72 p466~

〔25〕 J. Allen : "Anatomy of Lisp" 1978 (Mcgraw Hill)

〔26〕 K. Berkling : "Computer Architechture for Correct Programming" IE^3 1978 Computer Arehitechture

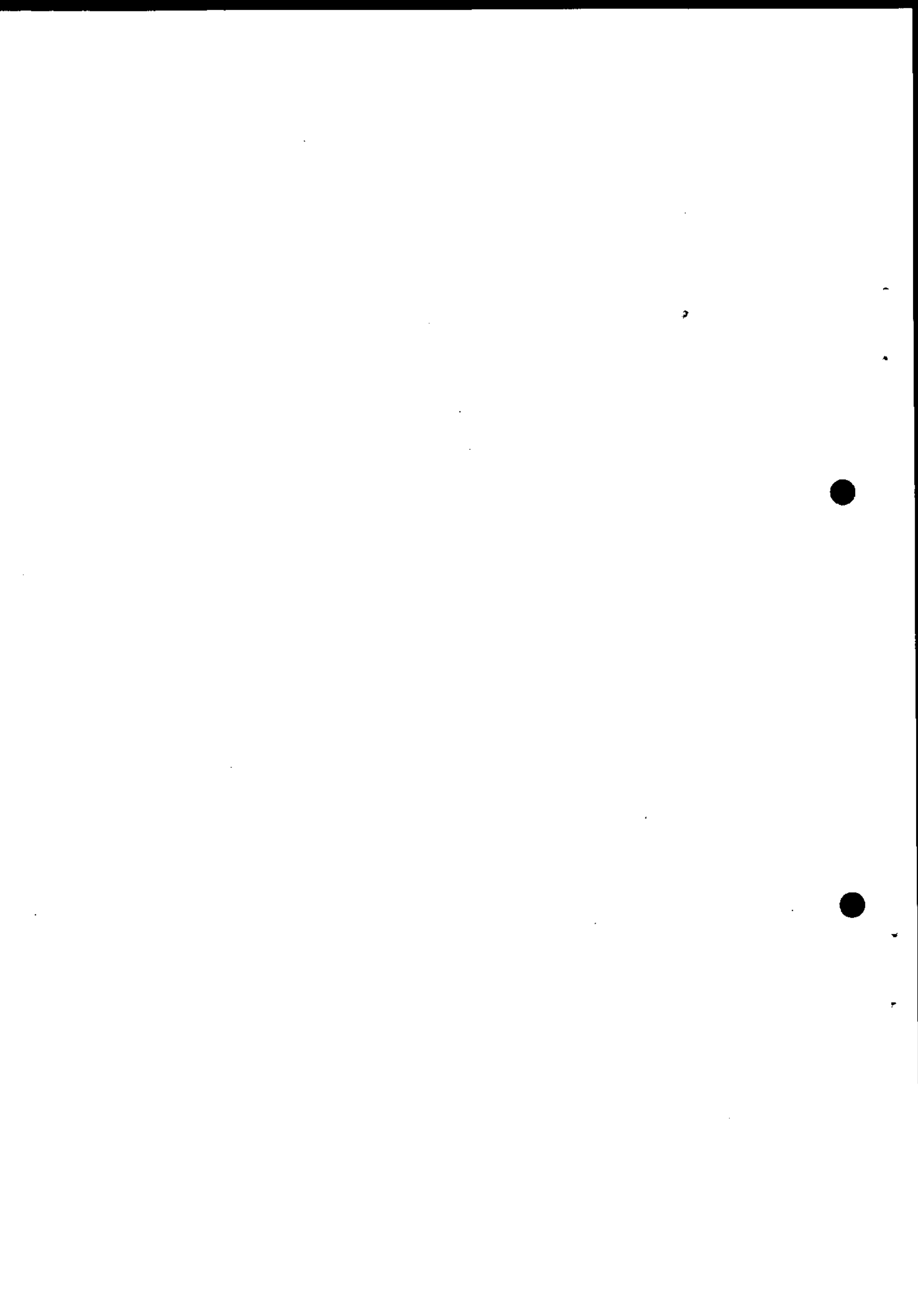
〔27〕 J. Backus : "Can programming be Liberated from the von Neumen Style? A Functional Style and its Algebra of Programms" ACM vol. 21 №8 1978

〔28〕 J. McCartly, Tom Binford, Cordell Green D. Luckham,

Z. Manna : "Recent Research in Artificial Intelligence and Foundations of Programming" Stanford AIM-321

[29] D. Harel : "First Order Dynamic Logic" Lec. Notes in Comp. Sci vol. 68

第４章　論理プログラミングについて
ー背景とその周辺，実現法ー



4. 論理プログラミングについて

4.1 はじめに

最近のコンピューターの性能（速度，容量）は素晴しい勢いで向上しているが，他方システム中のソフトのコストも素晴しい勢いで上昇している。原因として1つは巨大なプログラムを見通し良く作製する技術の不足，1つは“虫”のないプログラムを作る技術の不足が挙げられる。その為の対策として構造化プログラミング，abs ract-data type等々が提案されて来た。

ここで述べる論理プログラミングは“正しい”プログラムを見通し良く作製する技法であると同時に，コンピュータの利用のもう1つの形態 ― 計算の道具ではなく，知識を利用する道具としても有用なものである。

論理プログラミング（以下論理Pと略す）の考え方を紹介する手初めに，“計算とは何か？”論理学の方から眺めてみよう。

たとえば階乗関数の値を計算するとしよう。その為ユーザは次のような階乗関数の定義

(A) $f(n) = \begin{cases} 1 \ ; n = 0 \\ n * f(n-1) \ ; n > 0 \end{cases}$

から，適当なプログラミング言語を用いて

(B) f(n) = if n = 0 then 1

else n * f(n−1)

のように関数 f を定義し，システムに与えた後，f(0)や f(3)の値をシステムに尋ねるとシステムは f(0)の時は1を，f(3)の時は6を値として返すだろう。

さて数学的に言えば，関数 f は1つの特殊な N × N 上の2項関係であり，定義(A)はその2項関係を定義しているものと言える。即ち

(C) 関数 f $\Longleftrightarrow$ $\{ (n, n!) \} n \in N \}$

関数 f に付随する2項関係（f のグラフという）に F という名前を付けると，

n を与えて f(n)を求めることは

(D)　(n, m) ∈ F であるような m を探す

という事と等しい。論理Pではこの事態を次のように理解する。

定義(A)より，f のグラフ F については

$$
\text{(E)}\quad \begin{cases} F(0,\ 1) \\ \forall n \forall m\ (F(n,\ m * n) \leftarrow F(n-1,\ m)) \end{cases}
$$

但し，F(x, y)は(x, y)∈Fを示す。

が成立する。ユーザは論理的表現(E)＝論理プログラム(これも論理Pで示す)を与えた後，例えば，3に対してfの値を求めたい時は

$$\text{(E)} \vdash \exists m\ \ F(3,\ m)$$

つまり，(E)を公理と考えて，そこから　∃F(3, m)が導びけるか否か尋ねると，システムは，(E)⊢F(3, 6)であることを見出してユーザに知らせる。

“ 6 ”は正しい値である。何故なら

$$\text{(E)} \vdash F(3,6)$$

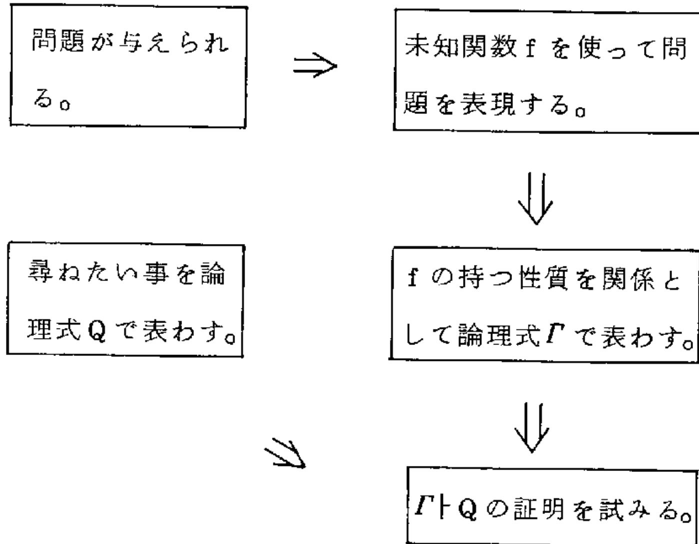
は，(E)をみたす任意の2項関係Fについて(3, 6)∈Fであることを意味する。そこでFとして，我々が考えた階乗関数のグラフをとるとそれは確かに公理(E)をみたし，従って

$$\text{(E)} \vdash F(3,\ 6) \Rightarrow (3,\ 6) \in \text{階乗関数のグラフ}$$

$$\Rightarrow 3\,! = 6$$

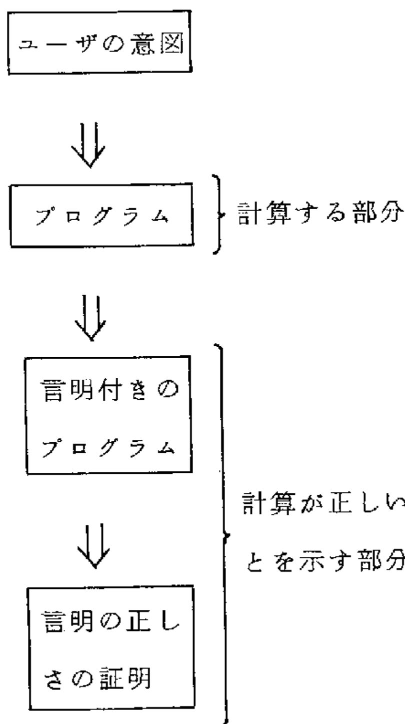
となるからである。

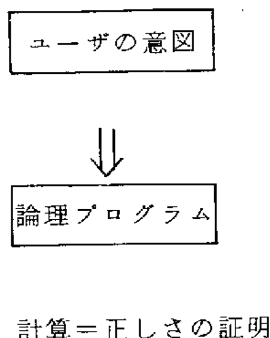
以上の取り扱いを一般化すると，論理Pのプログラム作製と利用の手順は関数の値を計算する場合次のようになる。



実は関数の値を計算するのみならず，知識を論理式で表現し，後でそれを利用する場合でも前記の手順は全く同一である。その場合異なることは，関数の時と違って，扱う関係が広いということである。

論理Pの原理的利点と難点を検討してみよう。利点の第1は明確な semantics を持っていることである。これは論理プログラムが与えられた段階でそれが何を表わしているか計算してみなくとも概念的に定まっているということである。semantics は通常の model theory が support してくれる。利点の第2は論理Pは始めから部分的正当性を持っていることである。つまり結果は常に正しい。" 正しい "という意味は，ユーザの意図が論理Pに正しく反映されているなら，言い換えると，書かれた論理Pがユーザの考えている model で満たされているなら，結果は常にユーザの model 上で成立するという意味である。従来の計算機言語では" 計算 "と" 計算の正しさの計算 "は異なるものであるが，論理Pでは両者は一体化している。





計算＝正しさの証明

論理Ｐの難点を考えてみよう。その最大のものは，制御構造の不足である。ユーザの意図は論理Ｐ＝公理Γと質問の論理式Ｑで表現できるが，$\Gamma \vdash Q$を試みる際，何をどうするといった証明過程の制御が（今のところ）できない。制御不足が論理Ｐの効率の悪さをもたらしがちである。従って制御構造を論理Ｐに導入する必要がある。この点については幾つかの提案が既にある。又論理Ｐの難点として，もし論理系の表現力が不足の時は問題を公理として表現できないということがある。たとえば副作用のある計算はどのように表現したら良いのであろうか？　これは新しい論理系の構築が必要になる可能性があることを示している。

次の節からは，論理系としてClausal Logicを選んで話を進める。このLogic（一階述語論理の部分系である）を選んだ理由は，syntax，semantics

の単純さと，にも拘わらず"計算可能な関数"はすべて表現できるからである。

4.2 Clausal Logic

4.2.1 Clausal Logicの導入

ここではClausal Logicを定義してその性質を探る。

≪Syntax≫

限量記号なしの一階述語論理式。例えば

$P(x, y) \wedge \neg Q(f(y), z)$, $R(x)$ 等々

≪semantics≫

変数はすべて∀がついているとして扱う。

≪推論規則≫

三段論法； $\vdash A$, $A \to B$ より $\vdash B$ を得る。

代　　入； $\vdash A$ より θ を代入として $\vdash A\theta$ を得る。

≪公　　理≫；命題論理の tautology にClausal Logicの formula を代入したもの，例えば $P(x) \to (Q(y) \to P(x))$,

$\neg\neg R(x, y) \to R(x, y)$ 等々

記述の便利の為記号"□"を導入する。"□"は $A \wedge \neg A$ の代用表現であり，矛盾を表わしている。

公理に加えて，式の集合 Γ を前提とした論理式Qの推論を $\Gamma \vdash Q$ と表わす。

通例の如く演　定理

- Rが閉式　$R, \Gamma \vdash Q \Rightarrow \Gamma \vdash R \to Q$

と完全性定理

- Qがすべてのモデルで真 $\Leftrightarrow$ $\vdash Q$

が成立する。又論理式Qが閉式（変数を含まない）時

- $\{\neg Q\} \cup \Gamma$ がモデルを持つ $\Leftrightarrow$ Not $\Gamma \vdash Q$

が成立する。結局 Clausal Logic で valid な式は tautology の instanciation に限ることがわかる。

4.2.2 Clausal Logic と Herbrand Universe

無矛盾な Clausal Logic の式の set はある特殊なタイプの model H-model(Herbrand model)を持つ。H-model は string から成る具体的な世界で計算機に馴み易いモデルである。

Γ の通常の model から H-model を作る話は標準的教科書に載っているので，ここでは前に 2.1 で述べたシステムを使って H-model を作る方法を示す。この項は後程述べる Horn set の最小 model の話とも関連する。

まず用語を定義する。

- リテラル≙素式(atomic fornwla)，又はそれに否定記号" "を前置したもの。
- 正(負)リテラル≙否定記号を持たない(持つ)リテラル。
- 閉リテラル≙変数なしのリテラル。

ついでに言うと有限個のリテラルを O R で結んだものを Clause と言う。

- Γ の H-univ.(Herbrand universe) $D_H(\Gamma)$≙
  次の手順で構成される string set
  H_0≙ 中の常数記号，なければ{" a "}
  $H_{k+1} \triangleq H_k \cup \{$" f "$(t_1, \cdots t_n) \mid$ " f "は n 一変数，$t_1 \cdots t_n \in H_K\}$
  $D_H(\Gamma) \triangleq \bigcup_{K=0}^{\infty} H_K$

- Γ の H-model $I_H(\Gamma)$≙
  領域が $D_H(\Gamma)$ であるような Γ の model

さて無矛盾な set Γ が与えられると次のようにして Γ の H-model を作ることができる。

Γ から生じる正(負)の閉リテラルの全体を $\alpha(\beta)$ で表わすことにする。

・ $\alpha \triangleq \{P(t_1 \cdots\cdots t_n)$ PはΓ中の述語記号,

$t_1 \cdots\cdots t_n \in D_H(\Gamma)\}$

・ $\beta \triangleq \{\neg P(t_1 \cdots\cdots t_n)$ PはΓ中の述語記号,

$t_1 \cdots\cdots t_n \in D_H(\Gamma)\}$

Γに対して$\alpha(\beta)$の部分集合δを無矛盾性を保ったまま付け加える，この時，付け加えることのできる極大な集合が存在する。何故なら

$\Gamma \cup \delta_n$　は無矛盾且つ$\delta_1 \subseteq \delta_2 \subseteq \cdots\cdots$とすると

$\Gamma \cup \bigcup_n \delta_n$　も無矛盾である。

従ってZornの補題より，$\Gamma \cup \delta$が無矛盾であるような極大のset　が存在することが言える。さて，δがそのような$\alpha(\beta)$の部分集合として，この時，任意の閉リテラルLについて

$\Gamma \cup \delta \vdash L$　又は$\Gamma \cup \delta \vdash \neg L$

が成立する。何故なら$\delta \subseteq \alpha$の時の，Lを負リテラルとして，

Not $\Gamma \cup \delta \vdash L \Rightarrow \Gamma \cup \delta \cup \{\neg L\}$ 無柔盾

$\Rightarrow$　δの極大性より$\delta \subseteq \delta \cup \{\neg L\}$

$\Rightarrow \Gamma \cup \delta \vdash \neg L$

となるからである。$\delta \subseteq \beta$の時も同様である。

δはΓを満たす解釈I_δを与える。

$I_\delta \models P(t_1 \cdots\cdots t_k)$　iff　$\Gamma \cup \delta \vdash P(t_1 \cdots\cdots t_k)$

とおくと$I_\delta \models \Gamma$である。即ちI_δはΓをみたす。これ任意の式$C \in \Gamma$と任意のground instanciation θ ($\theta = \{t_1/x_1 \cdots\cdots t_1/x_n\}$, $t_1 \cdots\cdots t_n \in D_H(\Gamma)$, $x_1 \cdots\cdots x_n$はC中の変数，の形)について，

$I_\delta \models C\theta$　iff　$\Gamma \cup \delta \vdash C\theta$

が証明できることから明らかである。

以上の手続きにより，無矛盾なClausal Logicの式のsetΓに対し，$\Gamma \cup \delta$が無矛盾且つ極大(maxmally consistent)な$\delta \subseteq \alpha(\beta)$を選ぶことにより，Γから生じるHerbrand universe $D_H(\Gamma)$上の解釈I_δを得ることができ

た。

ここでI_δ の意味を考えてみると，$\delta \subseteq \alpha$ つまり δ が正の閉リテラルから出きている場合，δ にこれ以上他の正の閉リテラルを加えると矛盾してしまうのであるから，I_δ は“なるべく多くの事が成り立つ解釈”である。逆に $\delta \subseteq \beta$ にとると，I_δ は“なるべく成り立つ事が少ない解釈”であることになる。しかし$\Gamma^U \delta$ がmaximally consistentになるset は数多く存在するので，“最も多くのことが成り立つ解釈”や“成り立つ事が最も少ない解釈”が存在するとは限らない。

4.2.3 Clausal Logicの拡張

次にClausal Logieの簡単な拡張を挙げよう。1つはsorts の導入と，if-then-else の導入であるが，実際はsyntax sugar として導入するのであり，semanticsはたいして変らない。

＊ many sortsの導入

各述語，関数，その他に各々固有のsort を持たせたい。例えば，“ 1 ”，“ 2 ”……等はNumberというsort を，“ aa ”，“ bb ”……等にはStringというsort を与え，1＋aa 等の表現を排除したい。

この為，項式(Well Formed Formula)の構成に於いて，各記号に対し，引数となる記号のsort を定めておき，項や式はsort の関係が正しいもののみを認めることにする。例えば，2引数関数“＋”の引数はnumeric symbol “ 0 ”，“ 1 ”，“ 2 ”……のみと定める。このようにした時4.2.1項で異なる所は，推論規則の代入の部分で$\vdash A$から$\vdash A\theta$を得る時，$A\theta$はsort に関し正しい式でなければならない。残りの部分はすべて同一である。

他のsorts の導入法もある。それはsorts predicates を導入して，例えば，人の先祖関係を，引数としてsort “ 人 ”を取る述語 先祖（$\underset{人}{\bullet}$，$\underset{人}{\bullet}$）を使って

$$x は y の先祖 \Leftrightarrow 先祖(\underset{人}{x}, \underset{人}{y})$$

と表わす代りに，述語“人”を使って

$$人(x)\wedge 人(y) \rightarrow 先祖(x, y)$$

と表わす方法である。この方法だと sorts 関の関係も表わすことができる。例えば

$$人(x) \rightarrow 動物(x),\quad \neg(人(x)\wedge 猿(x))$$

等を sorts に関する公理として付け加えればよい。この時，無矛盾ならば，実際に sorts の公理をみたす領域の存在が保証される。

sorts の話は，Resolution にも同様に導入できるが，紙面の都合上省略する。

＊ if－then－else の導入

項として if-then-else が使われる時，

(イ) $P(\cdots\cdots, \text{if } Q \text{ then } t_1 \text{ else } t_2, \cdots\cdots) \Leftrightarrow$

$$Q \rightarrow P(\cdots\cdots, t_1, \cdots\cdots) \quad Q \rightarrow P(\cdots\cdots, t_2, \cdots\cdots)$$

である。これを公理として採用すると，

(ロ) $\neg P(\cdots\cdots, \text{if } Q \text{ then } t_1 \text{ else } t_2, \cdots\cdots) \Leftrightarrow$

$$Q \rightarrow \neg P(\cdots\cdots, t_1 \cdots\cdots)\wedge \neg Q \rightarrow \neg P(\cdots\cdots, t_2, \cdots\cdots)$$

(ハ) $(P\wedge R)(\cdots\cdots, \text{if } Q \text{ then } t_1 \text{ else } t_2, \cdots\cdots) \Leftrightarrow$

$$Q \rightarrow (P\wedge R)(\cdots\cdots, t_1, \cdots\cdots)\wedge \neg Q \rightarrow (P\wedge R)(\cdots\cdots, t_1, \cdots\cdots)$$

が成立し結局，任意の Clausal formula の式 P について(イ)が成立する。通常の一階述語論理に if-then-else を導入するともう少し事情は複雑になるが，いずれにしても syntax sugar と考えてよい。

4.3 Horn set

論理式によるプログラミングを考えた時，実は clausal-Logic の部分系 Horn-Logic で充分である。Horn-Logic とは論理式の形を Horn に制限したものであり，効率の良い計算方法とある意味で一意的な model を提供してく

れる。

4.3.1 Hornの一般的定義とmodel-theory

一階述語論理に於けるHorn式(formula)は次のように定義される。

- $L_1 \vee \cdots\cdots \vee L_m$ 但し高々1つが正リテラルはHorn式である。

  P, QがHorn式の時,

- $P \wedge Q$, $\forall x Q$, $\exists x P$もHorn式である。

変数を含まないHorn式をHorn文(sentence)と言うことにする。Horn文は際立ったmodel theory的特徴を持つ。

model theory的特徴

- Horn文φのmodelのreduced productsは又φのmodelである。(連続体仮説のもとに逆も真である)
- 従ってHorn文φのmodelのdirect productも又φのmodelになる。

さてClausal Logicで扱うHorn式はUniversal Horn式である。このクラスをmodel的に特徴付けるのは容易で,即ち

- 一階述語論理式φが,Universal horn $\Leftrightarrow$

  φのmodelのsub-structureがmodel且つφの有限個のモデルのdirect-productが又modelになる。

- 従ってφのmodelの領域Dに対し任意の部分領域D′上にφの最小model* が存在する。

この事とClausal Logieの式のH-modelを考えると,horn set Γに

---

* 最小という意味は,φ中の任意の述語記号P, Q, R…に対し,解釈によって割り当てられる集合が最小という意味である。その意味で最小解釈とも言うことにする。

対し，D_H（Γ）上の最小modelが存在することがわかる。以後，式，文は，Clausal logicの式＝文を意味する。

horn 文をプログラミング言語として使うことの一つの意味はこのmodel的特徴にある。即ちUSERがhorn set Γ をシステムに提示した時，システムの持っているある領域D上に対し一意的にD上のΓの最小modelが定まる。これはΓが1つのmodel，関係を定義しそれがシステムに伝わることを意味する。即ちhorn 文Γは何かを定義してくれるわけである。他方一般の文では，そのmodelは一意的には（一般的には簡単なものを除いて）定まらない，つまりたとえUSERが一般の式のsetΓをシステムに与えても，システムの理解を越える言語による特別な指定がない限り，Γの特定のD上のmodel（Γが無矛盾な限りD上に必ず存在する）をidentifyすることができない。この事は一般的な文によっては何かを一意的に定義することができない事を示している。勿論システムに証明を行なわせるという意味ではUSER が一般的な文Γをシステムに与えることは構わない。唯その時はΓはstring以上のものではないわけである。

Clausal Logicに於ける最小modelを持つ式を少し眺めてみよう。horn setΓは最小のH-model を持つが，果して最小のH-model を持つものは，horn に限るであろうか？

Clausal Logicの無矛盾な式の集合Γは必ずDから生じるHerbrand領域D_H（Γ）上のmodel を持つ，D_H（Γ）を固定した時，model は各閉リテラルに対する解釈を与えれば決定される。そこで，簡便な方法として解釈Iを正リテラルの集合Iと考える。Iに入っていない正リテラルは，解釈Iによって偽であると考える。

$$I \models P \quad \text{iff} \quad P \in I$$

D_H（Γ）上の最小modelとは，集合としてIが最小であるような解釈である。解釈Iを集合Iとみると，D_H（Γ）上に最小解釈を持つsetΓは次のように特徴付けられる。

- ΓはD$_H$（Γ）上の最小解釈Iを持つ $\Leftrightarrow$ I≙{P| $\Gamma\vdash$P，且つPは正閉リテラル}，I $\models\Gamma$

つまりΓの論理的帰結によりΓのmodelを構成できる。即ちΓがrecursiveな場合，D$_H$（Γ）上に定義される関係はrecursive enumerableである。つまりrecursive modelを持つ。これは計算機で扱えるmodelという意味で相応しいmodelである。ΓがD$_H$（Γ）上の最小解釈を持つ為の必要充分条件は上記の他に

- if $\Gamma\vdash P_1\vee\cdots\cdots\vee Pn$ then $\exists i$ $\Gamma\vdash Pi$

  但し $P_1\cdots\cdots Pn$は正閉リテラル

とも表わせる。

命題論理に於ける最小解釈については，

- Γを命題論理の式の集合とする。この時Γが最小解釈Iを持つ。

  Γは$I^{\cup}J$と同値，但し

  I≙{P| $\Gamma\vdash$ P且つPはatom}

  J≙{Q| $\Gamma\vdash$ Q且つQ中にIのatomがない}

従って例えばΓ＝{A，B，C→D∨E}は最小解釈I＝{A，B}を持つ。他方ΓはHorn と同値ではない。何故なら

{A，B，C，D}＝$I_1\models\Gamma$，且つ{A，B，C，E}＝$I_2\models\Gamma$

しかし

{A，B，C}＝$I_1\cap I_2\not\models\Gamma$

であるからこれより

- Horn→最小解釈を持つ。しかし逆は真でない。即ち最小解釈を持つ文のクラスはHorn文のクラスより真に大である。

4.3.2 充足不能なHorn set Γに対するRefutation method

horn set ΓによりUSERがシステムの提供する領域D（当然すべての元には名前がついてる……一種のHerbrand領域である）上に関係Pを最小model

として定義したとする。USERの目的は，Pに対して質問Q（閉式とする）を発し，Pについての情報を引き出すことである。これは$\Gamma \vdash Q$? を尋ねることでもある。

これは$\Gamma \cup \{\neg Q\} \vdash \square$（矛盾），つまり$\Gamma \cup \{\neg Q\}$が充足不能である事と同値である。そこで充足不能なhorn set Γに対しresolution 法により□を導出する（refutation）ことを考える。充足不能なhorn set Γ に対するretutation は例えばSNL，SPU，LUSH等などのmethodがあるが，これらはすべてrefutation の図がtree になることかつ導びかれる。以下充足不能なhorn set Γのretutation method について幾つかの考察を示す。

- Γを閉式から成る極小充足不能※なhorn setとする。すると
  - (i) 正閉リテラルu がΓに出現しているならそれは唯1回に限る。
  - (ii) 少なくとも1つの負閉リテラル－u が唯1個Γ中に出現している。

※ 文のset Γが極小充足不能　$\Gamma = \{A_1, \cdots, Am\}$ とおくと，Γは充足不能且つ，Γから1つでもAi を除くと充足可能になる。

(iii) 従って，$\Gamma = \{uA_1, -uB_1, C_1, \cdots\cdots, Ck\}$ の形であり（A_1，B_1，C_1，……，Ckは適当なClause）且つ$\Gamma' = \{A_1, B_1, C_1 \cdots, Ck\}$も極小充足不能である。　$\Gamma' = \Gamma$ ※－1である。

(iv) 従ってΓに対し次のrefutation wethodがある。

(a) $\Gamma_0 \triangleq \Gamma$

(b) Γi 中に唯1回の出現を持つ負リテラル－u を持つClause $-uB_1$ を探し，これを正リテラルuを持つClause uA_1とresolveし $\Gamma_{i+1} \triangleq \Gamma_i - \{-uB_1\} - \{uA_1\} \cup \{A_1, B_1\}$

(c) $\Gamma i+1$ 中に□があればstop，なければ，i をi+1にして(b)へ

この方法は，$|\Gamma|$ −1回のresolutionでstopする。これ未満のresolution回数ではrefuteできないから，最も能率の良い方法である。**

さて上記の方法はΓが閉式の時は具合いが良いが，Γが変数を含むhorn setの時は能率が良いという保証はない。（証明をgeneral levellにlift-up する時，Γに於けるu，−uの唯1回の出現と言った情報が失なわれるのが原因である），Γが一般のhorn setでも適用できる能率の良いrefutation 法を探す基礎となるのは次の定理である。

- Γを充足不能な閉式から成る（＝ground levelの）horn setとする。ΓにP1 refutation (Positive hyper refutation)を適用すると次のことがわかる。

(i) factoringなし（P1 deductionのfactoringはpositive clauseのみに行なわれる。）

(ii) 証明の形はnegative clauseをtopとする木の形になる。

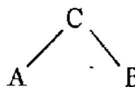
(iii) negative clauseは1つだけ証明に使われる。

例を挙げると解り易い

$\Gamma \triangleq \{A \wedge B \rightarrow C,\ D \wedge E \rightarrow F,\ C \wedge F \rightarrow G,\ \neg G$

$A,\ B,\ D,\ E\}$

は充足不能でその“証明の木”は左の図の形になる。例えば



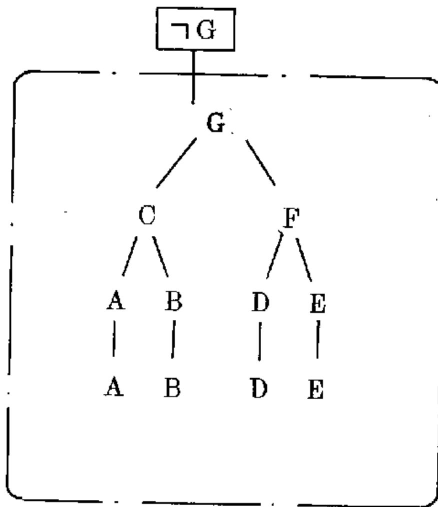
---

* $|A|$ はset Aの濃度を示す。

** $\Gamma = \{\bar{P}Q,\ P,\ \bar{Q}R,\ \bar{R}S,\ \bar{R}\bar{S}\bar{T},\ \bar{R}T\}$は5回のresolutionで□が出る。SickelのCIGによる方法でも10回程かかる。

Cf Special Issue on Automated Theorem Proving

IEEE, Trans. on C. Vol.25 №8 1976



は，A，Bがdeduce され，Γ中のclause A∧B→CによりCがdeduce されたことを示す。

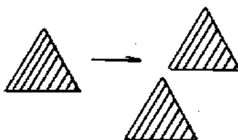
$\overset{|}{A}$ はAがΓ中に存在することを示す。

¬G はGがdeduce され，これとΓ中のnegative clause ¬Gがresolve されて□がdeduce されたことを示す。

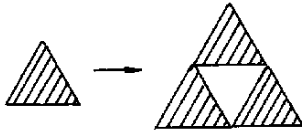
結局重要なのは—·—で囲った部分をΓから構成する方法である。（これはΓをContext Free ruleのset とみることを示唆する。
例えばA∧B→CをCF rule とみるとC→ABである。すると前の図は，開始記号をGとした文ABDE の導出であり，unit clause A，B，D，Eが語に相当する。しかし類推はここまでである。）証明の木はどこから作ってもより，又証明の木は複数あり得る。そこで木の作り方をContext Free grammarのparsing algorithmの用語を借りて分類する。

- top - down (I) SNLに対応

stackにより実働化される。



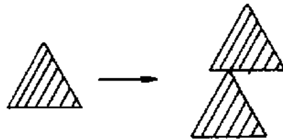
• top - down (II)



この方法には未だ名前がついていない。＊

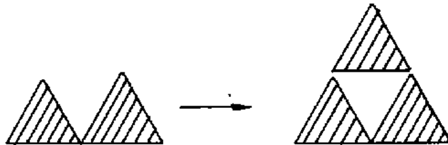
○ queue により実働化される。

• top - down (III)



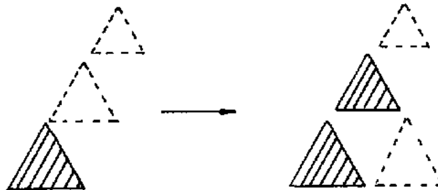
LUSH に対応，任意の node から下方へ span する。stack により実働化される。

• boffom - up (I)



SPU に対応

• boffom - up (II)　　　Earley の algorithm に対応



これにも名前がついていない，一応 Earley 式としておく

＊ cf 〔sickel〕 Uoriable Rauge Restnction in Resolution Theorew Proving, Machine Intelligence で提出されている。

- serial vs parallel

  1度に1つの証明の木を作る — serial法

  1度に全部の証明の木を作る — parallel法

証明法は，

$$\left\{\begin{array}{l}\text{top - down}\\\text{bottom-up}\end{array}\right\} \text{と} \left\{\begin{array}{l}\text{serial}\\\text{parallel}\end{array}\right\}$$

の組み合わせにより多数ある。実際はこれに制御構造を入れる。実例をあげると

- Top-down（Ⅰ）＋serial ＋ back-trackの制御

  ＝PROLOG

- Bottom-up（Ⅰ）＋serial ＋ confliction の制御

  ＝PSG，OPS等のProduction System

となっている。

論理Pの言語システムを考えると，システムの提供する証明法はなるべく多様な方が好ましい。その意味でTop-down（Ⅱ）やboffom-up（Ⅱ）の証明法も実働化すべきである。Top-down（Ⅱ）の実働化は大して問題がない（と思われる）が，boffom-up（Ⅱ）の実働化はそのままでは不可能である。何故ならCFGに於けるEarleyのparsing algorithm が働く為にはCFGrule の数が有限でなければならないが，一方，論理Pの各clauseは変数を含む場合無限コのCFGルールを表わしていると考えられるからである。例えば，P(x)∧Q（x）→R（x）はN上で｛P（φ）・Q（φ）→R（φ），P（1）・Q（1）→R（1），……………｝なるCFGルールのsetを示していると考えられる。この困難は，しかし，ある種の正則言語を導入することにより乗り越えられる。この言語については次節で，論理Pの“Compile” の概念と伴に説明する。

4.4 論理プログラムのコンパイル

論理Pをそのまま例えばPROLOG式に実行すると同じようなunitication を何度も繰り返すという意味で非能率である。そこで論理Pが与えられ

た段階でunificationの情報を取り出し(綜合グラフ),又証明の道筋を正則言語で表わし(論理Pからのプログラムの自動合成),実行時には,正則言語による証明の道筋に従って,前に取り出したunitication情報を利用しつつunifier を作ることにする。このやり方は,前節に述べた証明法を壊さず,各証明法の実行,クニックとして適用化能なものである。又停止性の証明にも有用である。

4.4.1 正則hornとcompile

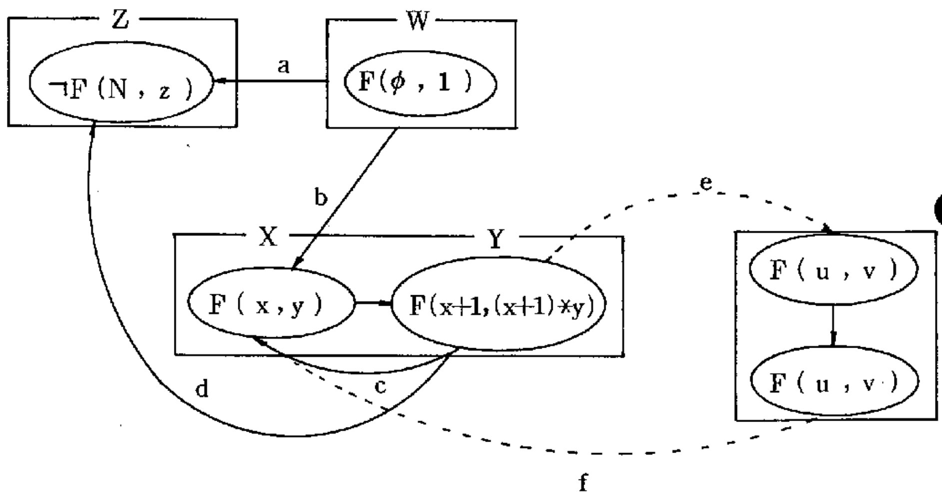
clauseの形がA, A→B, ¬B, のいずれかであるものを正則horn と呼ぶ。(A, Bは正リテラル)正則horn に於けるcompile を例をもって示す。

- 階乗関数の論理Pとcompileの例

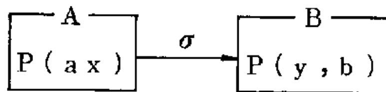
(イ) 論理P Γ
- 公理: F(φ, 1), F(x, y)→F(x+1, (x+1)*y),
- 質問: ¬F(N, z) Nはgiven constant

Γが与えられるとその結合グラフを作る。結合グラフとはΓ中のリテラルとリテラルをm・g・uで結んだ図である。

(ロ) Γの結合グラフ: C(Γ)



a, b, c, d, e, f はm・g・uである。m・g・u を等式の有限集合とみることにする。しかも等式の左図と右辺を区別する。例えば



の時は $\sigma = \{ b/x,\ a/y \}$ であるが，これを $\sigma = \{ a=y,\ x=b \}$ と考える。つまり矢印の出る方のリテラルの項を等式の左辺に，入る方のリテラルの項を右辺に書く

(ト) C (Γ)から生じる正則方程式，E (G) ; C (Γ)中のリテラルに固有名を付け，方程式E (Γ)を立てる。e , f はとりあえず無視する。＊

$$E(\Gamma):\quad \begin{cases} Z = W\ ;\ a + Y\ ;\ d \\ Y = X \\ X = W\ ;\ b + Y\ ;\ c \\ W = \varepsilon \end{cases}$$

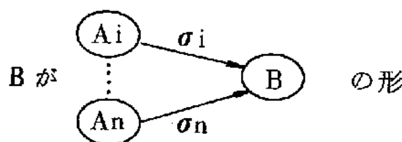
方程式の立て方の一般論は，C (Γ) 中で，

B が unit ⇒ B = ε

B が Ⓐ → Ⓑ の形　B = A

＊ e と f は変数の名前替えを自動的に行う為入れた clause F (u, υ) → F (u, υ) は tantology なので論理的に無害である。

B が



の形

$$\Rightarrow \quad B = A_1\ ;\ \sigma_1 \ + \cdots\cdots + A n\ ;\ \sigma\, n$$

である。

次に$E(\Gamma)$を，各unitier σ，及びεを語とし，正則方程式として解く，ここでは最小解を取ろう。

$$\begin{cases} Z = \varepsilon \; ; \; a + \varepsilon \; ; \; b \; ; \; c* \; ; \; d \\ X = Y = \varepsilon \; ; \; b \; ; \; c* \\ W = E \end{cases}$$

ここでは詳しく述べないが

- $C(\Gamma)$の解はΓのH-modelを生み出す。特に$E(\Gamma)$の最小解はΓの最小H-modelに対応する。

さて以上で階乗関数の論理Pからのcompileは終いである。質問clauseに対応するプログラムは

$$Z = \varepsilon \; ; \begin{Bmatrix} \phi = N \\ 1 = z \end{Bmatrix} + \varepsilon \; ; \begin{Bmatrix} \phi = x \\ 1 = y \end{Bmatrix} \; ; \begin{Bmatrix} x+1 = x \\ (x+1)*y = y \end{Bmatrix} * ; \begin{Bmatrix} x+1 = N \\ (x+1)*y = z \end{Bmatrix}$$

である。これをεをstart記号として左から右各unifierを順に実行*するとiteration，右から左へ実行すると(tail) recursionになる。各unifierの意味は

- $\begin{Bmatrix} \phi = N \\ 1 = z \end{Bmatrix} \Longleftrightarrow$ if $N = \phi$ then $z = 1$
  (iteration)

  or

  $\Longleftrightarrow$ if $N = \phi$ return 1
  (recursion)

何故ならNは常数，zは出力変数であるから。$\begin{Bmatrix} \phi = x \\ 1 = y \end{Bmatrix}$も同様。

- $\begin{Bmatrix} x+1 = x \\ (x+1)*y = y \end{Bmatrix} \Longleftrightarrow \langle x, y \rangle := \langle x+1, (x+1)*y \rangle$
  (iteration)

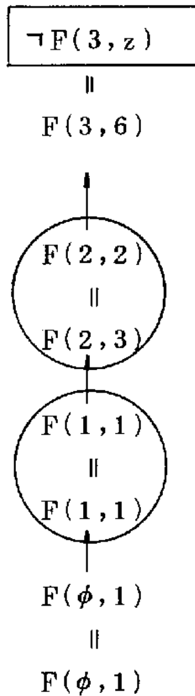
  or

* unifier の実行とは 中の等式をみたしてやることである。

y=y(x)と考えて

$$\Leftrightarrow \begin{cases} \text{if } y(x) \text{ is called then} \\ \text{return } x*y(x-1) \end{cases}$$

となる。Zを詳しく解析すれば，iteration (recursion)の回数はN回であることがわかる。従って証明の木は，N＝3の時，左図のようになる。Zの左から右への実行は左図の下から上への" parsing "Zの右から左への実行は，左図の上への" parsing "に対応する。



プログラムZは正しい，つまり，Zの実行後のNとZの関係は z＝N! である。これは，Dynamic Logic 風に書いて

$\Gamma \vdash$ 〔Z〕 F(N, z)

であることからわかる。Zの停止性の問題はZ中の＊の部分が有限になることの証明に帰着される。これは例えばNに関する帰納法によればよい。(一般に論理Pが働く世界は; H-universe なので，すべてのdata は帰納的に生成されており，data に関する帰納法が有用である。)

unifier σ，start(end)記号 ε，及び，；，＊，＋，から構成される前述のようなプログラムを言語Σ＊のプログラムと考える。

Σは primitive 操作として unification を取る＝unitier　σ中の等式を

＊　Σの syntax, senantics はここでは述べないが，大体 Dynamic Logic に沿っている。

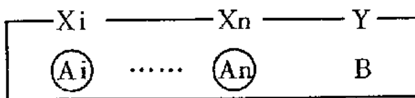
みたすというのを持ち，ε＝start，(end)　＋＝分枝，；＝順序実行，＊＝繰り返し，〔X〕＝手続きXの開始……Subroution callにより，大きなプログラムを作り出す。

次の節では一般horn の場合言語上にCompile する方法を考える。落ちたプログラムの左から右への実行が実は前節のbottom-up（Ⅱ）の実行法……Earley式に対応する。

4.4.2　一般hornのΣへのcompile

(イ)　充足不能なhorn setΓが当えられたとする。Γ中のnegutive clauseは1つ　ANS(x)であるとする。ANSをtopとする証明の木の言語Σによる記述を求める為，Γの結合グラフC(Γ)を作る。

(ロ)　C(Γ)中の各リテラルに相異なる名前を付け，方程式E(Γ)を立てる。
但し



の形は

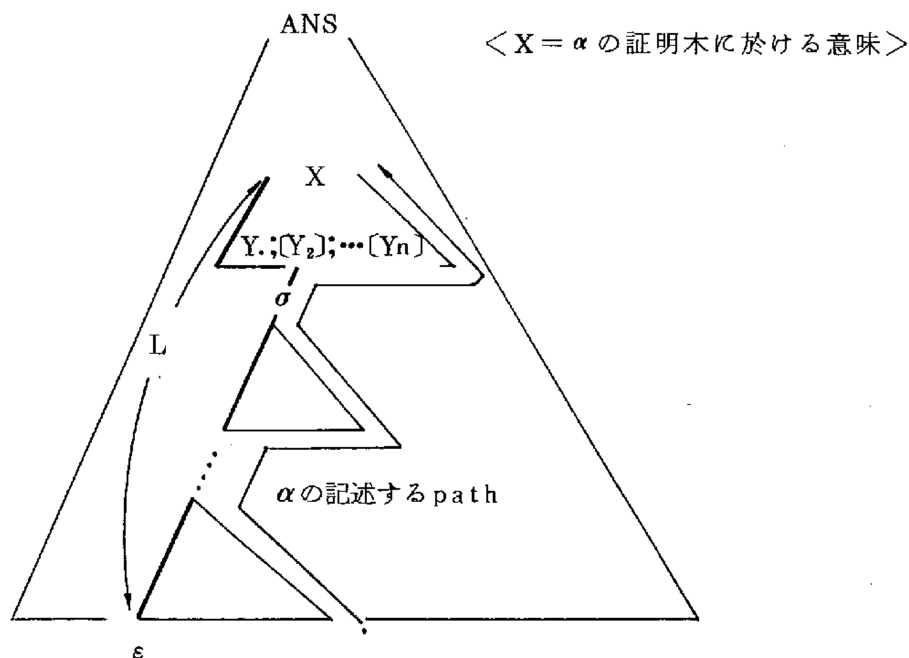
$Y = X_1$ ；〔X_2〕；……；〔Xn〕

とする。その他はregular horn の時と同じである。この式の意味は，
Earley式に実行される場合
状態X_1に達っしたら，X_2，……Xnのプログラムにより状態をチェックしてOKだったら状態Yへ行くということである。

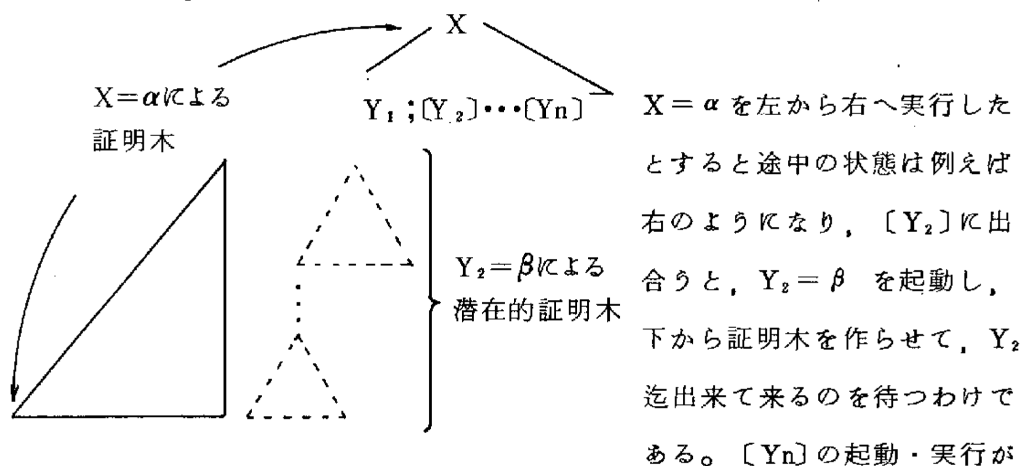
(ハ)　E(Γ)を正則方程式として解く，但し〔X〕等は1つの語として扱う。
予想によるとE(Γ)の任意の解はΓのH-modelに対応するはずである。
さて正則方程式E(Γ)の最小解は常に求まる。

(ニ)　ANS述語のラベルをXとし，X＝αが解であるなら，αがANSの言語　のcompiled version である。

E(Γ)の最小解で，例えば名前Xの解がαであったとする。これは証明の木があったとして，その中でXを頂点とする可能な部分証明木の左斜辺の全体に対応する。



もしX＝α中の〔・〕を無視するとそれは上の図でpath Lの部分の記述になっている。



X＝αを左から右へ実行したとすると途中の状態は例えば右のようになり，〔Y_2〕に出合うと，Y_2＝β を起動し，下から証明木を作らせて，Y_2迄出来て来るのを待つわけである。〔Yn〕の起動・実行がすむと，Xを top node とする部分証明木が完成するわけである。

以上でHorn set のEarley式証明法の説明を終える。なお付け加えると，

合成されたΣのプログラムは単一割り当て規則に従うので，そのdata-flow式実行法も興味ある問題である。

4.5 一般のclauseによる論理プログラミング

A→B∨C等のclauseを使った論理プログラミング（以下論理Pと略す）を実現したい。又その時，Horn setによる論理Pの実現と両立させたい。そこで場合分けの原理（Case Analysis, 以下CAと記す）を導入することによりSNLやEarley式による論理Pが可能なことを示す。

まず基本的idea を紹介する。

$$A\vee B\vee C,\ \Gamma\vdash\Box \iff \begin{array}{l} A,\ \Gamma\vdash\Box \\ B,\ \Gamma\vdash\Box \\ C,\ \Gamma\vdash\Box \end{array}$$

in elausal Logic
但し，A，B，Cは閉式

そこで上の図から

$$\begin{array}{l} A\vee \boxed{B}\vee \boxed{C},\ \ \Gamma\vdash \boxed{B}\vee C \\ \quad B\vee \boxed{C},\ \ \Gamma\vdash \boxed{C} \\ \qquad C,\ \ \Gamma\vdash\Box \end{array}$$

がわかる。但し $\boxed{B}$ ， $\boxed{C}$ は“無視”されていることを示す。無視されている式は恰もそれがないかのように扱われ，例えば $\boxed{B}\vee\boxed{C}$ のように無視された式だけになると始めてその無視を解かれB∨ $\boxed{C}$ の形で証明に参加する。

一般的には次のようにする。Γ中のclauseでnon-Horn $A_1\wedge\cdots\wedge Am\rightarrow B_1\vee\cdots\vee Bn$ があれば $A_1\wedge\cdots\wedge Am\rightarrow B_1\vee\boxed{B_2}\cdots\boxed{Bn}$ の形に□で余計なリテラルをcloseする。各non-horn clauseに対し□による余計なリテラルのcosingを終えてから□のついたリテラル——closedリテラルに通し番号を添る。

これで準備はOKで，あとはSNLやSPU又はEarley式にrefutationを試みればよい，途中で例えば $A_2\vee B_4\vee C_5\vee A_6$ 等のclosedリテラル

のみから成るclauseが出てきたら，その番号の最小のものを(non-deterministic　)選び，openし($A \vee \boxed{B}_4 \vee \boxed{C}_5 \vee \boxed{A}_6$ になる)，openリテラルについてmergeする($A \vee \boxed{B}_4 \vee \boxed{C}_5$　)，これを恰もinput clause のようにして証明に参加せる。unifier σはcloseしたリテラルにも作用させる。

Closeするのは何もリテラルに限らない。例えば，$A \wedge B \rightarrow C \vee D$の時，$A \rightarrow C \vee \boxed{B \rightarrow D}$ の形にclose してもよい，あとの手順は全く前述の方法と同じである。この場合，$\boxed{B \rightarrow D}$ という補題が証明され，使われることになる。

- CAはOL，Semantic，Earley式，すべての方法とCompatibleである。即ち完全性を保つ。

CAにより，non-Horn set ΓにもΣへのCompilationが可能になるが，詳細は略す。

4.6　おわりに

以上論理Pの背景や一般化，具体化への説明を試みた。論理Pは何もPROLOGだけが唯一の道ではないこと，又論理Pが様々に拡張可能であることが理解されたと思う。

論理Pにはまだまだ解くべき問題が残っている。例えばglobal変数はどう実現されるのか？　これらは未解決の問題として大きく論理Pの前方に立ち塞っているものである。

紙面の都合上，論理Pによる計算の側面のみ語って来たが，論理Pは知識の表現，利用の問題，即ちData-Baseとも深い関係を持っている。この辺りのことは筆者の手に余ることなので，是非とも成著なり，このレポート集の関連レポートに当ってみて欲しい。

〈参考文献〉

〔渕　76〕渕一博“定理証明として見たEarley／PraHのパージング・アルゴリズム”情報処理第17回全国大会（1976）

〔渕　77〕渕一博“述語論理的プロブラミングーEPILOGの提案ー”情報処理一記号処理研究会（1977）

〔渕　78〕渕一博“述語論理の中での仕様からプログラムへの変換”情報処理第19回全国大会（1978）

〔渕　79〕渕一博“述語論理的プログラムと他言語との関連”情報処理第20回全国大会（1979）

〔佐藤　80〕佐藤泰介“Logic Programming, Parsing, Compilation”情報処理第21回全国大会（1980）……… 発表予定

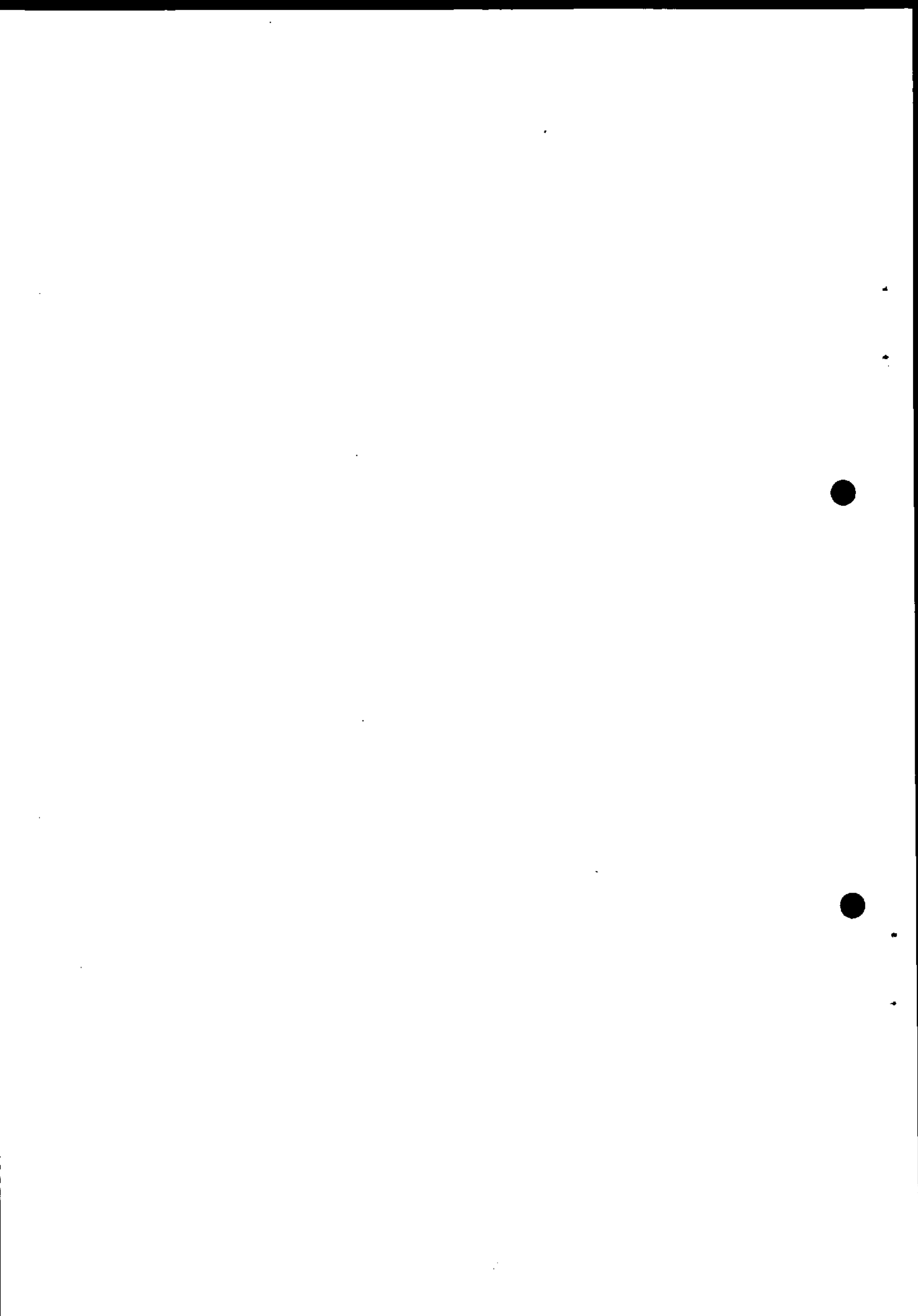
〔Kanoui 76〕Kanoui, H“Some Aspects of Symbolic Integration via Prodicate Logic Programming”SIGSAM Vol. 10 №4（№V. 1976）

〔Warren 77〕Warren, D. H. D. and Pereiva, L. M.“PROLOG -The Lauquage and its Inplementation Compared with Lisp,”SIGPLAN, Vol. 12, №8（Aug. 1977）

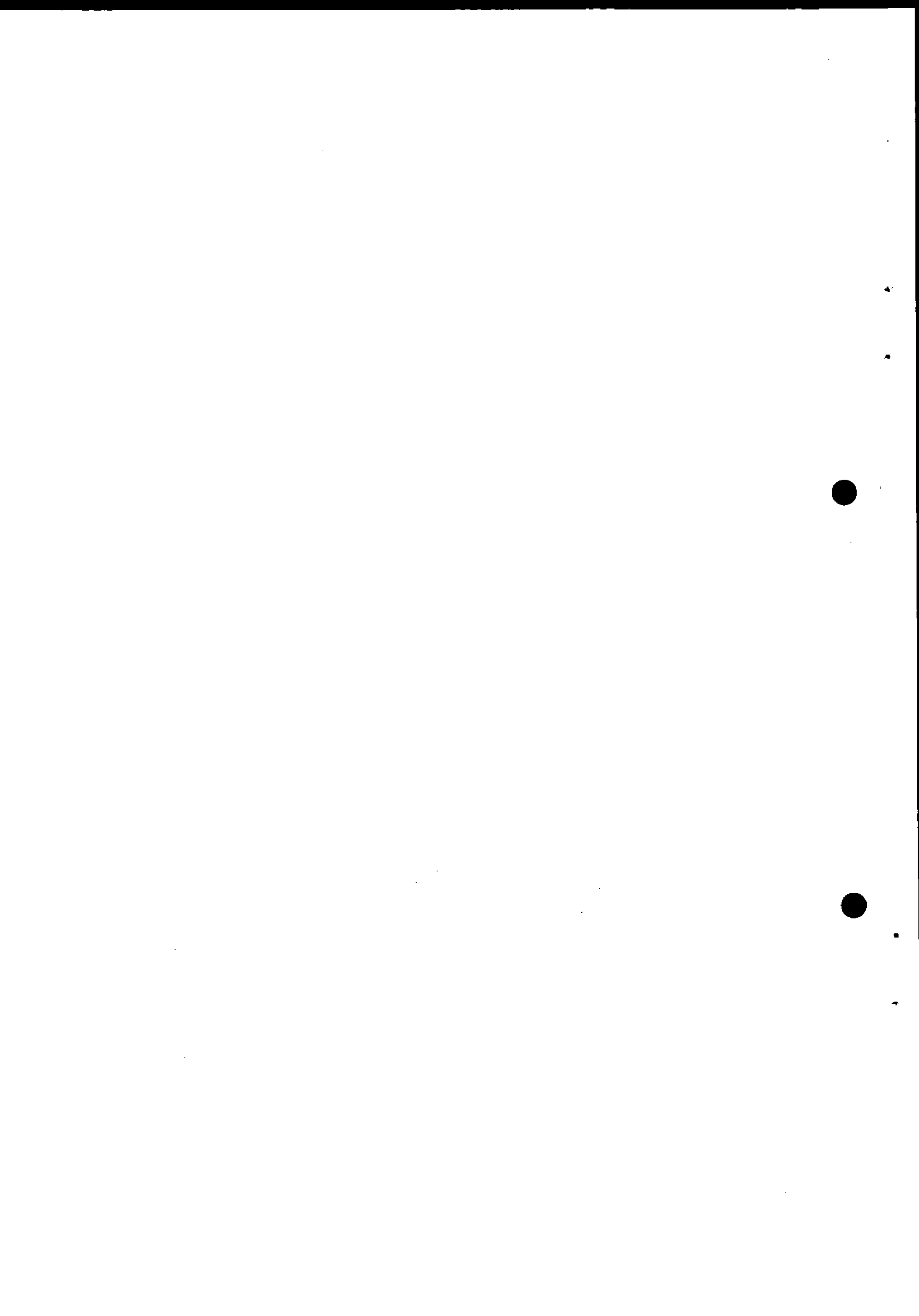
〔Clark 77〕 Clark, K. and Sickel. S " Predicate Logic : A Cal culus for Deriving Programs, " 5th IJCAI (Aug. 1977)

〔Sickel 77〕 Suckel, S" Variable Range Restrictions in Resolution Theovem Proving " Machine Intelligerce 8, ELLIS HOKWOOD Ltd. England, (1977)

〔Kowalski 79〕 Kowalski, R. " Algorithm=Logic+Control " CACM Vol. 22 №7 (July 1979)



第5章　代数的プログラミング
－代数的仕様とその階層的プログラミングへの応用－



5. 代数的プログラミング

5.1 まえがき

“良いプログラム”が備えるべき特質としては，

- reliability（信頼性）
- understandability（わかり易さ）
- efficiency（効率）
- modifiability（変更し易さ）

などがある。計算機技術の発展とともに，プログラムの巨大化，複雑化が進み上記の性質の内，reliability（信頼性）とunderstandability（わかり易さ）が特に重要視されるようになった。その結果，プログラムの構造の重要性が認識されるようになったといえる〔11〕。理想的には，プログラムは独立性の高い要素プログラム群へ分解される（decompose）べきであり，その構造は次のような条件を満たすことが望まれる。

(1) そのプログラムによって解かれるべき問題の構造を忠実に反映している。

(2) そのプログラムによって引き起こされる計算の構造を忠実に反映している。

プログラムを，計算機で解くべき現実の問題とその問題を計算機上で実際に解く計算の間の interface と考えれば，上記(1)，(2)の条件は，この interfaceを通じての現実の問題の構造と計算機上の計算の構造との整合が取れていることを意味する。プログラムにこのような条件を満たす良い構造を持たせるためには，問題の構造及び計算の構造がどのようなものであるかを明らかにしなければならないだろう。

計算は，実際に動いている計算機（hardware)を基礎にして，その構造を比較的厳密に述べることが可能である。具体的には，計算の構造は通常計算機アーキテクチャとして与えられる。現状計算機においては，instruction co-

unter と memory cell の集合体とを中心とするいわゆる"Von Neumann 型"の構造がその主流となっている。

一方，問題の構造がどのようなものであるかを述べるのは容易でない。計算機を用いて機械的に解かれるべき，または解くことが可能である問題の構造，というのが最も一般的な記述であろうが，これではその詳細は明らかでない。問題の構造に対する簡単な記述を与えられない最大の理由は，それが，計算の理論，プログラムの理論，計算機の理論，プログラミング方法論といった計算機科学の総体の上に築き上げられるべき概念的なものであることによる。従って計算機科学の進展に伴って問題の構造も変容を続け，そのあるべき姿を捉えることは非常に困難である。我々に出来ることは，現時点での計算機科学の現状を踏えて問題の構造を的確に反映したモデルを提出し，そのモデルを検証していく作業を行なうことである。このようなモデル作りの基礎として注目すべきものに，"抽象化に基づく構造化"の考え方がある。これは，抽象度の高い問題の理解から出発して，徐々に抽象度を下げながら問題の理解を詳細化してゆき，最終的には，問題を抽象化のレベルに従った階層構造を持ったものとして捉えるという考え方である。これは人間がある問題を何らかの意味で機械処理しようとする際の非常に基本的な原理であり，この"抽象化のレベルに基づく階層構造"をプログラム化されるべき問題の基本構造と考えるのは妥当であろう。

上述のように，計算の構造として現状の Von Neumann 型の構造を，計算機で解かれるべき問題の基本構造として"抽象化のレベルに基づく階層構造"をとるとすれば，プログラミング方法論，計算機アーキテクチャの分野で現在見られる次のような動きは，すべてこの計算と問題の構造の不整合を取り除こうとする方向を目指していると見なすことが出来る。

(a) 代数的，論理的，関数型などと称される，新しいプログラミング形態を求める動き。

(b) Data flow machine, Reduction Machine など新しい計算の構造

を実現しようとする動き。

本稿ではこの内，問題に内在する〝抽象化のレベルに基づく階層構造″を直接反映したプログラムを作り上げるための最有力な道具と思われる，代数に基づくプログラミングのモデル化について述べる。

5.2 代数的仕様

5.2.1 なぜ代数か？

プログラム化すべき問題の基本構造を，5.1 で述べたように，抽象化のレベルに基づく階層構造として捉えるとしても，問題あるいはその階層的記述をどのようにモデル化して具体的な表現を与えればよいか（仕様記述をすればよいか）は明らかでない。この〝問題のモデル″（これを逆から見れば〝プログラムのモデル″でもある）として現在最も有望と思われるのに〝代数的仕様″がある。これは本来プログラミングにおける抽象データ型の仕様として考え出されたものであるが，その後この考え方を拡張してプログラミング問題一般のモデル（仕様）として利用しようとする動きに発展しつつある〔1〕，〔2〕，〔9〕，〔10〕，〔8〕。

抽象データ型は，ある特定の要素の集まりの上に働く，互いに関連し合った複数の演算の集まりとして捉えられる。従って，要素の振舞いはそれら演算を適用することによってのみ知ることが出来る。このようにモデル化することにより，データ型をその実現法（データ構造やそれへのアクセス法）と独立に考えることが出来る利点（抽象化の利点）が生ずるのである。

抽象データ型の重要性は広く認められ，基礎から応用まで多くの研究が進行中である〔3〕，〔4〕，〔5〕，〔7〕。
これらの研究の中でも最も基本的なものの一つとして，抽象データ型の仕様記述法に関するものがある。代数的仕様記述法はこのような仕様記述法の一つとして考案され，現在では多くの研究者がその有用性を認めつつあると言える。

（たとへば〔7〕を参照）

代数的仕様は，多ソート代数（many sorted algebra）を基礎とする。プログラミング問題のモデル化においてデータをいくつかのタイプに分類することが，それを基礎にしたプログラミングの reliability を上げることは，Pascal などタイプ化されたプログラミング言語の例を引くまでもないであろう。従って，問題のモデル化に当って" 多ソート "とすることについては異論は少ないと思われる。しかしながら，なぜ代数に限定して意味記述の手段として代数的等式（$term_1 = term_2$の形の公理）のみを使うのかという疑問は残る。これについては，次のような理由が挙げられよう。

(1) プログラミングの全過程，仕様記述（モデル化）からコーディング／実行に至るすべての処理を単一の形式化の中で行うことには，操作システムの簡易化など多くの利点が期待出来る。等式に基づく形式化はこの点に関しては現時点で最も有望視されるものである。（述語論理に基づく形式化では，Horn clause に限定した Prolog に代表される，Logic Programing が同様の利点を有すると思われる。）

(2) 等式による問題の仕様記述法は，λ−計算を基礎にしたプログラミング言語，特にＬＩＳＰ など，で蓄積された種々の技法を拡張又は流用することにより，比較的容易に整備，体系化出来る可能性が高い。

5.2.2 代数的仕様

以下では，Word algebra（Harbrand universe）のみを対象として議論を進める。この特殊な代数は，同じ" 型 "を持つ代数の中で initial である（この代数から他の代数へ unique な homomorphism が存在する）ことから，その代数的意味は一意に定まる（同型の意味で一意に定まる）という利点を持つ〔4〕。さらに，term の集合から構成される word algebra は，term がＬＩＳＰ のリスト構造などと同じ構造を持つことから，計算機処理を考える際に既存の多くの know howを使えるという利点も有する。

Sをsort名（タイプ名）の集合とする。S sorted signature Σは，$\langle w, s\rangle \in S^* \times S$を引数に持つ互いに素な集合の族

$$\Sigma = \langle \Sigma w, s \rangle \; w \in S^*, \; s \in S$$

である。ここで，$\Sigma w, s$は，arity $w \in S^*$，sort $s \in S$ の$\langle w, s\rangle$型のoperator記号の集合である。

Σ algebra Aは，carrier と呼ばれる$s \in S$ を引数に持つ集合の族$\langle A_s \rangle \; s \in S$と，$\Sigma$に属する各operator記号$\sigma \in \Sigma w, s$に割当てられた$\langle w, s\rangle$ 型のoperation σ_Aからなる。つまり，$w = s_1 \cdots\cdots s_n$ とすれば，

$$A : A_{s1} \times \cdot\cdot\cdot \times A_{s_n} \rightarrow A_s$$

変数記号集合$X = \langle X_s \rangle s \in S$ を，$s \in S$を引数に持つ互いに素な変数記号の集合の族とする。

今後，混乱のない限り$\Sigma = \cup w \in S^*, s \in S \; \Sigma w, s$，$X = \cup s \in S \; X_s$ として話を進める。

$\langle T_{\Sigma, s(X)} \rangle \; s \in S$ を次の条件を満たす$(\Sigma \cup \{(, \;)\})^*$の部分集合$T_{\Sigma, s(X)}$の族とする。（$\{(\; , \;)\}$は$\Sigma$と素とする。）

(1) $\Sigma_{\lambda, s} \; \cup (X s \cap X) \subseteq T_{\Sigma, s}$（$\lambda$は空系列）

(2) $\sigma \in \Sigma w, s$，$w = s_1 \cdots\cdots s_n$，$n > 0$かつ$t_i \in T_{\Sigma, s(X)}$ とすれば，

$$\sigma(t_1 \cdot\cdot\cdot tn) \in T_{\Sigma, s(X)}$$

(3) (1)，(2)で$T_{\Sigma, s(X)}$元とされるものだけが$T_{\Sigma, s(X)}$の元である。

$T_{\Sigma, s(X)}$をsort sのtermの集合という。

$$T_{\Sigma(X)} \stackrel{\text{def}}{=} \cup s \in S \; T_{\Sigma, s(X)}$$

とし，特にXが空のとき$T_{\Sigma, s(\phi)}$をT_s，$T_{\Sigma(\phi)} = T_\Sigma$と記す。

$T_{\Sigma, s(X)}$上の同値関係$\equiv s$の族$\langle \equiv s \rangle \; s \in S$は，$\sigma \in \Sigma_{s_1 \cdots\cdots s_n, s}$，$t_i$，$t_i{}' \in T_{\Sigma, s_i(X)}$に対し，もし任意のiに対し$t_i =_{s_i} t_i{}'$なら

$$\sigma(t_1 \cdot\cdot\cdot tn) \equiv s \;\; \sigma(t_1' \cdot\cdot\cdot tn')$$

が成立するとき，Σ-congruenceといわれる。

Σ-congruence $\equiv \stackrel{\text{def}}{=} \langle \equiv s \rangle \; s \in S$による商集合の族$T_{\Sigma(X)}/\equiv$は，

Σに属する各operatorに対し次のようなoperation $\sigma_{\equiv}$を割当てればΣ-algebraとなる。

(1) $\sigma \varepsilon \Sigma_{\lambda, s}$ なら $\sigma_{\equiv} = [\sigma]$

(2) $\sigma \varepsilon \Sigma_{s_1 \cdots s_n, s}$ かつ $[t_i] \varepsilon T_{\Sigma, s_i}(X)/\equiv_{s_i}$

ならば

$$\sigma_{\equiv}([t_1], \cdots [t_n]) = [\sigma(t_1 \cdots t_n)]$$

ここで，$[t]$ ($t \varepsilon T_{\Sigma(X)}$) は t を含む同値類を表わす。この$\Sigma$-algebraを$\equiv$が定める$\Sigma$-word algebraという。

Rを$T_{\Sigma, s}(X)$上の2項関係Rsの族$\langle Rs \rangle_{s \varepsilon S}$とする。Rを含む最小の$\Sigma$-congruence$\equiv_R$が存在する。これをRにより生成されるcongruence関係と呼ぶ。

次の条件を満たす変数集合Xから$T_{\Sigma(X)}$への写像θを"置き換え"と呼ぶ。

(1) $\theta(x) = x$ でないXの元は有限個しかない。

(2) $\theta(x) = t$ かつ$t \neq x$ならtとxのsortは一致している。(ある s S に対し，x，$t \varepsilon T_{\Sigma, s}(X)$。)

置き換えθの定義域は，

$$\theta(\sigma(t_1 \cdots t_n)) = \sigma(\theta(t_1) \cdots \theta(t_n))$$

と定義することにより，Xから$T_{\Sigma, s}(X)$へと拡張される。t_1が変数記号を含まない定数termであれば，

$$\theta(t_1) = t_1$$

となることに注意されたい。

同一sortのtermの組$e = (t_L, t_R)$をequationと呼ぶ。equationの集合Eに対し，Eから生成される$T_{\Sigma, s}(X)$上の2項関係の族$R(E) = \langle Rs(E) \rangle s \varepsilon S$を次のように定義する。$e \varepsilon E$で$e = (t_L, t_R)$かつ$t_L$，$t_R$のsortがsのとき，任意の置き換え$\theta$に対し，

$$(\theta(t_L), \theta(t_R)) \varepsilon Rs(E)$$

$R(E)$から生成されたΣ-congruenceを$\equiv_E$と記す。

以上の準備の基に代数的仕様を次のように定義することが出来る。

〔定　義〕代数的仕様は3字組（S，Σ，*E*）で定義される。ここで，Sはsort 名の有限集合，ΣはS-signatureであり，*E*はequationの集合である。

このように定義された代数的仕様に対し，その意味は*E*から生成されるΣ-word algebraとして定義される。

5.2.3　書き換え規則に基づく意味定義

5.2.2において，代数的仕様を3字組（S，Σ，*E*）とし，その意味は*E*が生成するΣ-congrueuce $\equiv_E$ が定めるΣ-word algebraとして与えられると定義した。これは代数的仕様の数学的に厳密な定義ではあるが，この定義のみに基づいて仕様記述を行なったり，仕様の意味を理解するのは容易ではない。従って代数的仕様に対するより構成的で直観的に理解し易い"意味の定義"が必要となる。ここでは，そのような意味の定義として，代数的公理を書き換え規則と見なす代数的仕様の意味定義法について述べる。これはまた代数的意味の具体的計算法を与えるものであり，代数的仕様を直接実行するアルゴリズムの基礎を提供している。後に述べるようにすべての代数的仕様の意味を書き換え規則に基づいて定義出来るわけではない。しかしながら，仕様記述　プログラミングにおいて重要な概念の多くが書き換え規則に基づいて定義可能である。代数的仕様の多くに対し，書き換え規則に基づいた簡単な意味定義が可能であり，それがそのままその仕様の実現法を提供していることが，書き換え規則に基づく意味定義の最大の利点である。

equationは次のようにして書き換え規則とみなされる。

Σ-term $t_1 \in T_{\Sigma}(X)$ は，ある置き換え θ とΣ-term $t_2 \in T_{\Sigma}(X)$ が存在して，$\theta(t_2) = t_1$ となるとき t_2型であるともいわれる。*E*をequationの集合とする。Σ-term tは，あるequation $e = (t_L, t_R) \in E$ とtの部分term（t自身もtの部分termとする）t'が存在して，t' が t_L 型である

ときEに関してreducibleであるともいわれる。θを$\theta(t_L) = t'$である置き換えとし，uをtの部分term t'を$\theta(t_R)$で置き換えて得られるtermとする。このときtはEを用いてuに直接reduceされるといわれ，

$$t \xrightarrow{E} u$$

と記す。$\xrightarrow{E}$は$T_{\Sigma(X)}$上の2項関係を定義する。この2項関係を$Q(E)$と記す。定義から$Q(E)$は$T_{\Sigma,s(X)}$上の2項関係$Qs(E)$の族とみせることに注意されたい。さらに，先に5.2.2で定義した$R(E)$と$Q(E)$との相違にも注意されたい。

$Q^*(E)$と$\overline{Q}^*(E)$でそれぞれ，$Q(E)$の反射・推移閉包(reflexive-transitive closure)と反射・推移・対称閉包(reflexive-trnsitive-symmctric closure)を表わすとする。

$\overline{Q}^*(E)$は同値関係であり，定義より，Eから生成されるΣ-congruence $\equiv_E$に一致する。

$(t\ \ u) \varepsilon Q^*(E)$かつ$(t\ \ u') \varepsilon Q^*(E)$ならば，ある$\Sigma$-term $v \varepsilon T_{\Sigma(X)}$が存在して，$(u\ \ v) \varepsilon Q^*(E)$かつ$(u'\ \ v) \varepsilon Q^*(E)$となるとき，$Q(E)$はChurch-Rosserの性質を持つといわれる。$(u\ \ v) \varepsilon Q^*(E)$なるvが存在しないときuは$E$-reducedと呼ばれる。$(t\ \ u) \varepsilon \overline{Q}^*(E)$でuが$E$-reducedならば，uはtの$E$-reduced formであるといわれる。

$\overline{Q}^*(E)$が定める任意の同値類が高々1つのreduced formしか含まないとき，$Q(E)$はunique termination property(UTP)を持つという。

定義からただちに，もし$Q(E)$がChurch-Rosserの性質を持てばそれはまたUTPも持つことが分かる。

さらに次の命題が成立する。

〔命　題〕[3] もし$Q(E)$がChurch-Rosserの性質を持つならば，$(t\ \ u) \varepsilon \overline{Q}^*(E)$かつuがreducedであれば$(t\ \ u)\ \varepsilon Q^*(E)$が成立する。

この命題から，もし$Q(E)$がChurch-Rosserの性質を持てば，代数的仕様(S, Σ, E)の意味は，Eを書き換え規則とみなして順次適用することにより

求めることができることになる。ただし，Q(*E*)がChurch-Rosser の性質を持つだけでは，書き換え規則の適用が必ず終了する保証はないことに注意されたい。

equationの集合*E*に対しQ(*E*)がChurch-Rosserの性質を持つための条件を求める研究は多い〔6〕,〔2〕。ここでは次の定理を述べるにとどめる。

〔定　理〕 equationの集合*E*が次の性質を持てば，Q(*E*)はChurch-Rosser の性質を持つ。

(1) 任意の(t_L　t_R)ε *E*に対し，t_R中に現われる変数はすべてt_Lに現われる。

(2) 任意の(t_L　t_R)ε *E*に対し，t_L中に同じ変数が2度現われることはない。

(3) (t　u),(t'　u')ε *E*を2つの異なるequationとする。どんな置き換えθ，θ'に対しても$\theta(t)$は$\theta'(t')$の部分termにならない。

5.3 代数的仕様に基づく階層的仕様記述／プログラミング法

この節では，現在筆者らが研究開発中の階層的仕様記述／プログラミシグ・システムHISP(HIerarchically Structured Specification and/or Program Processor)の基礎となる仕様記述／プログラミング法について述べる。ここで，仕様記述／プログラミングという術語を使っているのは，我々の方法では仕様記述とプログラミングは同質のものであり，単にその抽象化の程度に差があるだけであると考えているからである。この意味において以後は仕様記述とプログラミングは区別せず，当面は抽象化の度合の高い記述を問題にしたいということから仕様記述という術語を主に使うことにする。

プログラミング問題の仕様記述を極く単純に，operatorの集合の階層構造を系統的に構成することとしてモデル化するのが基本的な着想である。このモデルでは，唯一の仕様記述の単位はobjectと呼ばれ，次のように定義される。

objectは4字組

$\langle$ SUB, S, O, E $\rangle$

で表現される。ここで，ＳＵＢはsub-objectのＳはsortの，Ｏはoperatorの，Ｅはequationの，それぞれ有限集合である。ＳＵＢは単にobjectの集合であり空でもよい。

objectの有限集合ＯＢは，次の条件を満たすとき階層構造を持つという。

HOB1：任意の$ob \varepsilon OB$に対し，

$ob = \langle$ SUB, S, O, E $\rangle$

とすると，$SUB \subseteq OB$である。

HOB2：任意の$ob_\phi \varepsilon OB$に対し，

$\forall i \varepsilon \{\phi, 1, \cdots, n-1\}$：　$obi \varepsilon SUB_{i+1}$かつ$ob_n \varepsilon SUB\phi$

であるようなobjectの系列

$ob_i = \langle SUB_i, S_i, O_i, E_i \rangle$

$(i = \phi, 1, 2, \cdots, n)$

は存在しない。

条件ＨＯＢ２はobjectの循還的な定義を禁止している。ただし，二進木などの再帰構造の定義はsortとoperatorのレベルで可能なことに注意されたい。

階層構造を持つobjectの集合ＯＢは，それに属する任意のobject

$ob = \langle \{ob_1, ob_2, \cdots, ob_k\}, S, O, E \rangle$

$ob_i = \langle SUB_i, S_i, O_i, E_i \rangle$

$(i = 1, 2, \cdots, k)$

が次の条件を満たすとき，局所化されているという。

LOB1：Ｏに属するoperatorの定義域または値域になっているsort はすべて

$S \cup S_1 \cup S_2 \cup \cdots \cup S_K$

の要素である。

LOB2：Eに属するequationの左辺または右辺のtermを構成するoperatorはすべて

$$O \cup O_1 \cup O_2 \cup \cdots \cup O_k$$

の要素である。

以上の準備のもとに，仕様記述は次のようにモデル化される。つまり，

仕様とは，階層構造を持つ局所化されたobjectの集合であり，この集合を系統的に構成することが仕様記述である。

objectの階層構造として定義された仕様は次のようにして代数的仕様とみなされる。

OBを仕様，つまり階層構造を持つ局所化されたobjectの集合とする。OBの任意の元

$$ob = \langle \{ ob_1, ob_2, \cdots, ob_n \}, S, O, E \rangle$$

に対して，repS，repΣ，repE を次のように再帰的に定義する。

(1) $repS(ob) = S \cup \cup_{i=1}^{n} repS(ob_i)$

(2) $rep\Sigma(ob) = O \cup \cup_{i=1}^{n} rep\Sigma(ob_i)$

(3) $repE(ob) = E \cup \cup_{i=1}^{n} repE(ob_i)$

任意のobject $ob \in OB$ に対し，それに対応する代数的仕様を定める関数repは，

$$rep(ob) = (repS(ob), rep\Sigma(ob), repE(ob))$$

で与えられる。

階層的仕様（objectの集合）から定められる代数的仕様はさらに以下のような性質を持つことが要求される。

2つの代数的仕様 $D_1 = (S_1, \Sigma_1, E_1)$，$D_2 = (S_2, \Sigma_2, E_2)$ に対し，$S_1 \supseteq S_2$，$\Sigma_1 \supseteq \Sigma_2$，$E_1 \supseteq E_2$ であるとき，D_2 を D_1 の部分仕様と呼ぶ。さらに，すべての t，$t' \in T_{\Sigma_2}(X)$ に対し，$t \equiv_{E_1} t'$ ならば $t \equiv_{E_2} t'$ が成立するとき D_2 は D_1 中でprotertされているという。

階層的仕様（objectの集合）OBに属する任意の元，

$ob = \langle \{ ob_1, ob_2, \cdots, ob_n \}, S, O, E \rangle$

に対し，それから定められる代数的仕様rep(ob)は明らかに，obのsub-object $ob_1, ob_2, \cdots, ob_n$ から定められる代数的仕様 $rep(ob_1), rep(ob_2), \cdots, rep(ob_n)$ を部分仕様として含む。これらの部分仕様はrep(ob)中で，protectされていなければならないとする。階層構造を持つ局所化された，objectの集合として定められた仕様は，この条件を満たして初めてその代数的仕様としての意味が定まるものと考える。以後objectの集合として定まる階層的仕様は，すべてこの条件を満たしているものとする。

以上のようにモデル化された仕様（objectの集合）は次の5つの基本演算を用いて構成される。

(i) Creation：sub-objectを持たないobjectを作る。

(ii) Construction：すでにあるobjectをsub-objectとして利用して新してobjectを作る。

(iii) Refinement：sub-object，equationなどを添加することによりすでにあるobjectを詳細化する。

(iv) Renaming：sort，operatorなどの名前換えをする。

(v) Substitution：sub-objectを他のobjectですげ換える。

HISP言語は，階層構造を持つ局所化されたobjectの集合を構成するための言語であり，上で導入した5つの基本演算に対応した5つの構文単位を持つ。ここでは，その5つの構文単位を簡単な例を用いて示す。

(i) Creation

```
BOOL ::   /*Boolean algebra*/
  create
    sort Bool
    op   true, false : -> Bool
         not_ : Bool -> Bool
         _ and _ : Bool, Bool -> Bool
         _ or _ : Bool, Bool -> Bool
```

```
    eq    var ?b? : Bool
          ( not true = false )
          ( not false = true )
          ( true and ?b? = ?b? )
          ( false and ?b? = false )
          ( true or ?b? = trne )
          ( false or ?b? = ?b? )
  end
```

(ii) Construction

```
ANY ::
  create sort Any end

STACK ::
  constr
    sub   BOOL, ANY
    sort  Stack
    op    nil : -> Stack
          push _ to _ : Any, Stack -> Stack
          empty? : Stack -> Bool
          pop : Stack -> Stack
          top-of _ : Stack -> Any
          underflow : -> Stack ( error )
          undef : -> Any ( error )
    eq    var ?a? : Any ; ?s? : Stack
          ( empty? ( nil ) = true )
          ( empty? ( push ?a? to ?s? ) = false )
          ( pop ( nil ) = underflow )
          ( pop ( push ?a? to ?s? ) = ?s? )
          ( top-of nil = undef )
          ( top-of ( push ?a? to ?s? ) = ?s? )
  end
```

(iii) Renaming and Substitution

```
INTSTACK ::   /* STACK of INTeger */
  STACK (* ANY <- INTEGER; Any <- Integer *)
```

```
(% Stack <- IntStack %)
```

(v) Refinement

```
STACKR :: /* STACK with Replace */
  refIne
    op replace : Stack, Any -> Stack
    eq var ?a? : Any ; ?s? :/ Stack
       ( replace (?s? ?a?) =
            push ?a? to pop(?s?) )
   end
```

上述の substitution 及び refinement 演算を用いることにより，仕様を top-down的に書き下すことが出来る〔2〕。 これがＨＩＳＰ言語の大きな特徴である。

＜参考文献＞

〔1〕 R. M. Burstall and J. A. Goguen, Putting theories together to make specifications, Proc. of 5th Int. Joint Conf. on Artificial Intelligence, MIT July 1977, 1045−1058.

〔2〕 K. Futatsugi and K. Okada, Specification Writing as Constructions of Hierarchically Structured Clusters of Operators, accepted for IFIP Congress 80, TOKYO

〔3〕 J. A. Goguen, Some design principles and theory for OBJ−O, A language to express and execute algebraic specifications of progams, Proc. of IFIP2.2 Conf., Kyoto, August 1978, 429−475.

〔4〕 J. A. Goguen, J. W. Thatcher and E. Wagner, An initial algebra approach to the specifieation, and implementation of abstract data types, Current trends in programming methodology, Vol. IV: Data structuring (R. T. Yeh, Ed.), prentice-Hall, 1978.

〔5〕 J. V. Guttag, The specification and application to programming abstract data types, Univ. of Tronto, CSRG−59, 1975.

〔6〕 M. J. O′Donnell, Computing in systems described by equations, Lecture Notes in Computer Science, Vol.58, Springer-Verlag, 1977.

〔７〕 Proceedings of of Specification of Reliable Software, IEEE Catalog №79 CH1401−C (1979)

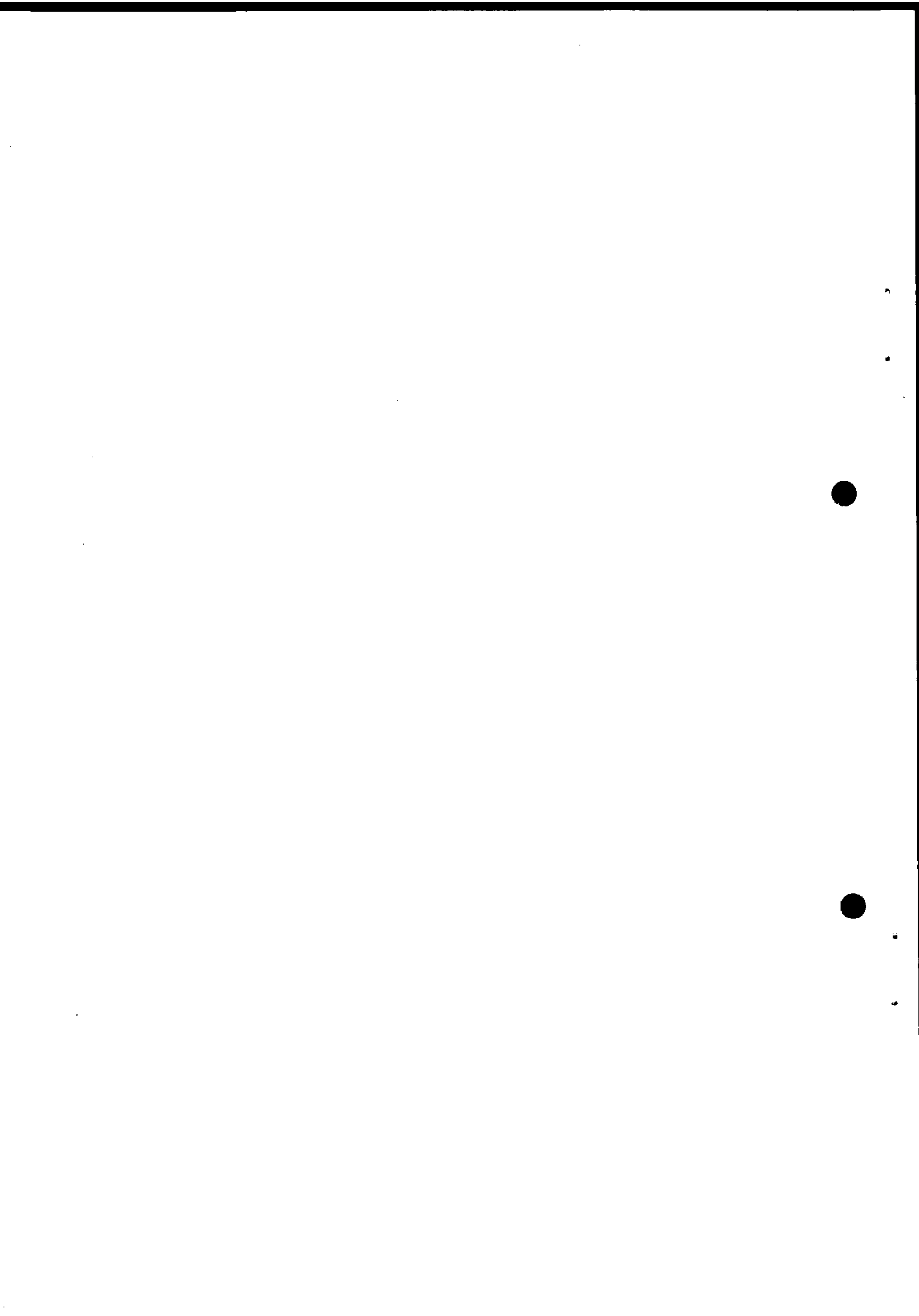
〔８〕 二木，岡田，階層的仕様記述の一モデル，情処学会２１回全大，２Ｃ−３，１９８０

〔９〕 嵩他，プログラム仕様記述の代数的方法について，信学会，技術報告ＡＬ７８−５，１９７８

〔10〕 岡田，二木，ＨＩＳＰシステムにおける階層的仕様記述，情処学会２０回全大，２Ｉ−６ １９７９

〔11〕 鳥居，二木，真野，プログラミング方法論の展望，情報処理，２０−１，ＰＰ２２−４３

第6章　数式とプログラム



6. 数式とプログラム

6.1 数式処理システム

計算機に行わせる内容が高度になるにつれ，扱うデータも，数値・文字といった単純なものから，それらを組合わせて構造を持たせたものを単位としたほうが便利となってくる。

数式処理はそのうちで，我々が通常使用する文字までを含んだ数式，たとえば $x+2$ や ax^2+bx+c という式，を一つのデータだと考えて処理を行なうものである。数式というものは純粋に数学的な概念であり，その変形規則も公理として明確な形で表現されている。たとえば

$$a+0=a$$

$$a\cdot 1=a$$

$$ab+ac=a(b+c)$$

などが公理である。

数式処理システムでは，これらの公理を書き換え規則として数式を求める形に変形するものである。数式どうしの演算，加減乗除算，さらには特定の変数についての微分や積分等，を計算機に行わせることができれば，今まで人間の手計算によるしかなかった式変形の操作を自動的に行なうことができるようになる〔16〕。

このような処理をまとめて一つのシステムとして作り上げたものが幾つか発表されている〔8〕〔10〕。（それらの代表的なもの，MACSYMA，REDUCEはともにLispで書かれており，数式をリスト構造で表現して処理を行なっている。）

ところで公理のなかには $a+0=a$，$a\cdot 1=a$ のように変形の方向が一意に決まるもの（この場合は「左辺→右辺」）と $ab+ac=a(b+c)$ のように一意には決まらないものがある。そのため一般の数式についてはそれの標準形（見かけは異なる式でも，一定の操作をほどこすことによってある一つの式

に変形できるならばその式）が存在しないことが，証明されている〔9〕〔15〕〔17〕。このため，実際のシステムでは，扱える式の範囲を限定するか，不完全なままで処理を進めることになる〔11〕〔12〕。また数式処理では式変形の途中で誤差が入ってはいけないので，浮動小数点よりも有理数で計算をすることが多くなる。するとすぐに扱う数の範囲が現在の計算機の一語で表現できるところをこえてしまうために，任意多倍長整数の演算が必要である。

このような数式処理システムを，理論面・作成面・利用面の三つの面から図式化すると図6-1のようになる。この図の左上方の不定形の部分は数式処理

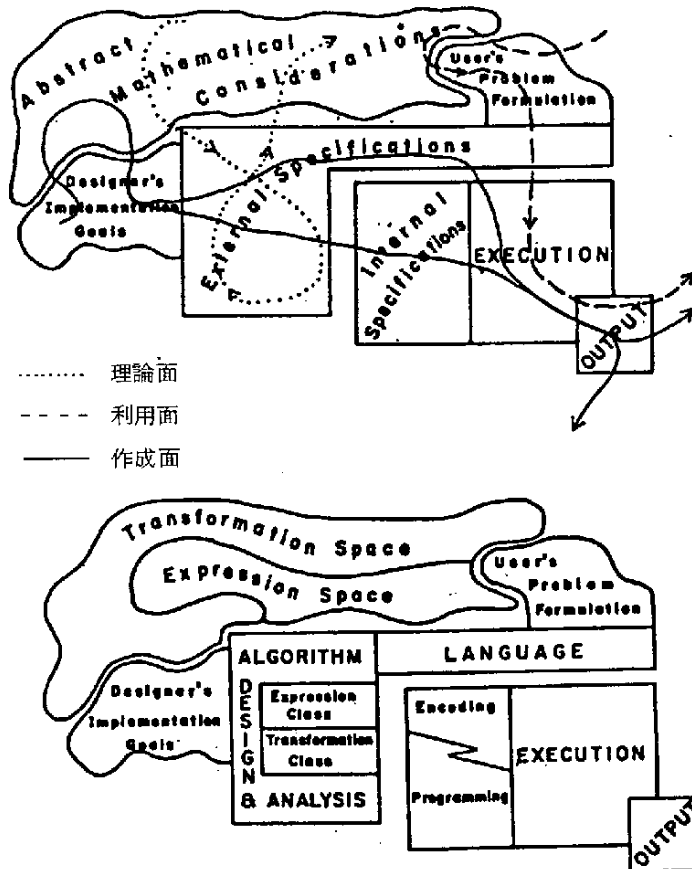


図6－1　数式処理システムのモデル
(R.G.Tobey:Symbolic Mathematical Computation ─ Introduction and Overview)

で実現しようと目ざしている数学的概念であり，実際のシステムではその部分集合が実現されて，中央部のTransformation ClassとExpression Classを構成している。この二つのClassは実際の概念の部分集合であるために，利用者が問題を定式化する際の制限となることがあるので，十分な注意が必要である。さらに利用者が定式化をプログラムするためにプログラム言語も必要となる。システムの作成者は，式変形の強力さと扱える式の豊かさを天秤にかける必要がある。というのは，アルゴリズムの設計や分析・そのシステムで使用できる式や変形規則・式の表現などをきめるときに，式変形に関する数学的な裏付けを考慮しなければならないからである。さらに作成者は，言語の設計や，デバッグをする時のために出力を行えるようにする必要もある。

6.2 代表的システム

6.2.1 MACSYMA

MACSYMAは米国MITのMAC ProjectでMoses 教授等を中心に開発されたシステムであり，それ以前のMATHLAB の後をうけてできたものである。

MACSYMAはMACLISP で書かれており，そのソースはラインプリンタ用紙で一千枚に及ぶほど大きなシステムである。現在はPDP−10あるいはMITのLisp machine上で動いているが，MACLISPをサポートしていない他の計算機上で動かすことは非常に困難である。

持っている機能は通常の加減乗除算・微分のほかに，RischのAlgorithmを取り入れた積分・極限あるいは不定積分を求めること・級数展開・微分方程式を解く等と豊富であり，他のシステムを一歩リードしている。また，MACSYMA は利用者が対話的に利用することを主目的にしており，そのために出力ルーチンも実際に数学で使用する表現に近い形で表示するように二次元表現となっており，分数などは分子・分母が線の上下に位置するようになっている。

6.2.2 REDUCE

REDUCEは現在ユタ大学にいるHearn教授が中心になって作成したシステムである。

REDUCEは移殖性をもたせることに重点が置かれており，大抵のLisp に備わっているかあるいは実現の容易な機能からなるStandard Lispで書かれている。このシステムはもとは理論物理の計算用に作られたものであり，移殖性があることと相まって，計算機関係以外の分野でも広く利用されているシステムである。機能の豊富さという点ではMACSYMA に一歩遅れをとるが，変形規則はかなり強力である。また物理関係への応用のために，各変数に重みをつけた近似展開・相対論のテンソル計算・スピンに関する計算が容易に行える。

6.3 数式の変形

与えられた数式を望む形に変形する過程を計算機で行う場合に，基本的な方法はどのシステムでも共通しているので詳しく見ることにする。

まず，我々が通常数学で使用する記号を計算機の内部表現に直す必要がある。内部の基本構造としてリストを用いることにし，処理し易いように，演算子は前置型で表現することにすると，たとえば

$$ax^2+bx+c$$

は

```
(PLUS (TIMES A (EXPT X 2))
      (TIMES B X)
      C)
```

に，

$$\int_0^2 \frac{\sin x}{x^2+1}\,dx$$

は

```
(INTEGRAL (QUOTIENT (SIN X)
                    (PLUS (EXPT X 2) 1))
          X 0 2)
```

というように，二次元的な表現を，入れ子構造を持ったリストで表わせる。実際のシステムでは二次元のものを入力するものは現在は無く，FORTRAN流に

A*X↑2+B*X+C

という書式で入力を行っている。

このようにしてできた内部表現を変形するわけであるが，以下の説明では式を内部表現のリストで書くと見にくいので，それと等価な通常の数学で使用する表現を用いることにする。

式の変形においては，最も演算の優先順位の高い内側から順位の低い外側に向かって実行していく。例として

$$a \cdot x \cdot x + 1 \cdot x + (b + 0) \cdot x + c - x \qquad (*)$$

を変形する場合を考える。

この場合に，プログラムも数式の一種と考えられるので，数式としての文字とプログラム変数としての文字との区別をどのようにするかという問題がある。この例では(*)式の変形を行なう前に

$$x \leftarrow y + 1$$

という文があってxの値が定義されているときに(*)式中のxとしては，ẋという文字そのものをとるか，y＋1というxの値をとるかという選択があることになる。数学では変形以前に値の定義された文字はその値を使用しており，現在，多くのシステムでも値のある変数はその値を使い，値のない場合はその文字自体を使うようになっている。ここでは後者の方法で行なうことにすると(*)式のxはy＋1で置きかえられ，a，b，cはその文字自体となって(*)式は

$$a \cdot (y+1) \cdot (y+1) + 1 \cdot (y+1) + (b+0) \cdot (y+1) + c - (y+1)$$

となる。

この式に変形を加えるわけであるが，1・(y+1)やb+0は公理によってそれぞれ(y+1)，bとなるので

$$a\cdot(y+1)\cdot(y+1)+(y+1)+b(y+1)+c-(y+1)$$

となり公理を使用し同類項をまとめると

$$a(y+1)^2+b(y+1)+c$$

となる。

ここまではどのようなシステムでも同様に行うが，これ以後は利用する目的によって望む形が違うことがあるので，一般的に論ずるのは困難になる。

また変形する式の中に

$$\int(3x-x)\cdot e^{x\cdot x}dx$$

のように関数（積分も一種の関数と考える。）がある場合には，その関数を適用して

$$e^{x^2}$$

とするか，適用しないで

$$\int 2xe^{x^2}dx$$

の形にするかを指定できるようなシステムもある。

6.4 プログラムと数式処理

プログラムをデータ操作に関する公理の集まりとして記述する方法が考えられる。この方法では，数式という数学的に明確な対象を扱うために，従来のプログラミング言語におけるような数学的におかしな解釈をしないでプログラムが書ける可能性がある。

その際に，今まで行なわれて来た数式処理の技術は大いに参考になると思われる。

＜参考文献＞

〔1〕 SIGSAM Bulletin; ACM special interest group on Symbolic and Algebraic Manipulation.

〔2〕 Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation; 1971.

〔3〕 Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation; 1976.

〔4〕 Proceedings of the 1977 MACSYMA Users' Conference; 1977.

〔5〕 Proceedings of the EUROSAM '79 (EUROpean symposium on Symbolic and Algebraic Manipulation); 1979.

〔6〕 MACSYMA Reference Manual; the MATHLAB Group, Project MAC-MIT, 1974.

〔7〕 REDUCE 2 User's Manual; A.C. Hearn, Univ. of Stanford, 1973.

〔8〕 BARTON, D. and FITCH, J.P.: A Review of Algebraic Manipulative Programs and Their Application, Comp. J. vol 15, pp362-381.

〔9〕 CAVINESS, B.F.: On Canonical Forms and Simplification, J. ACM vol 17, 1970, pp385-396.

〔10〕 KANADA, Y.; Implementation of HLISP and Algebraic Manipulation Language REDUCE 2, univ. of Tokyo, Information Science Lab. Tech. Rep. 75-01, 1975.

〔11〕 MARTIN, W.A.: Determining the Equivalence of Algebraic Expressions by Hash Coding, J. ACM vol 18, 1971, pp549-558.

〔12〕 MOSES, J.: Algebraic Simplification: A Guide for

the Perplexed, C. ACM vol 14, 1971, pp527−537.

〔13〕 MOSES, J.: Symbolic Integration: The Stormy, C. ACM vol 14, 1971, pp548−560.

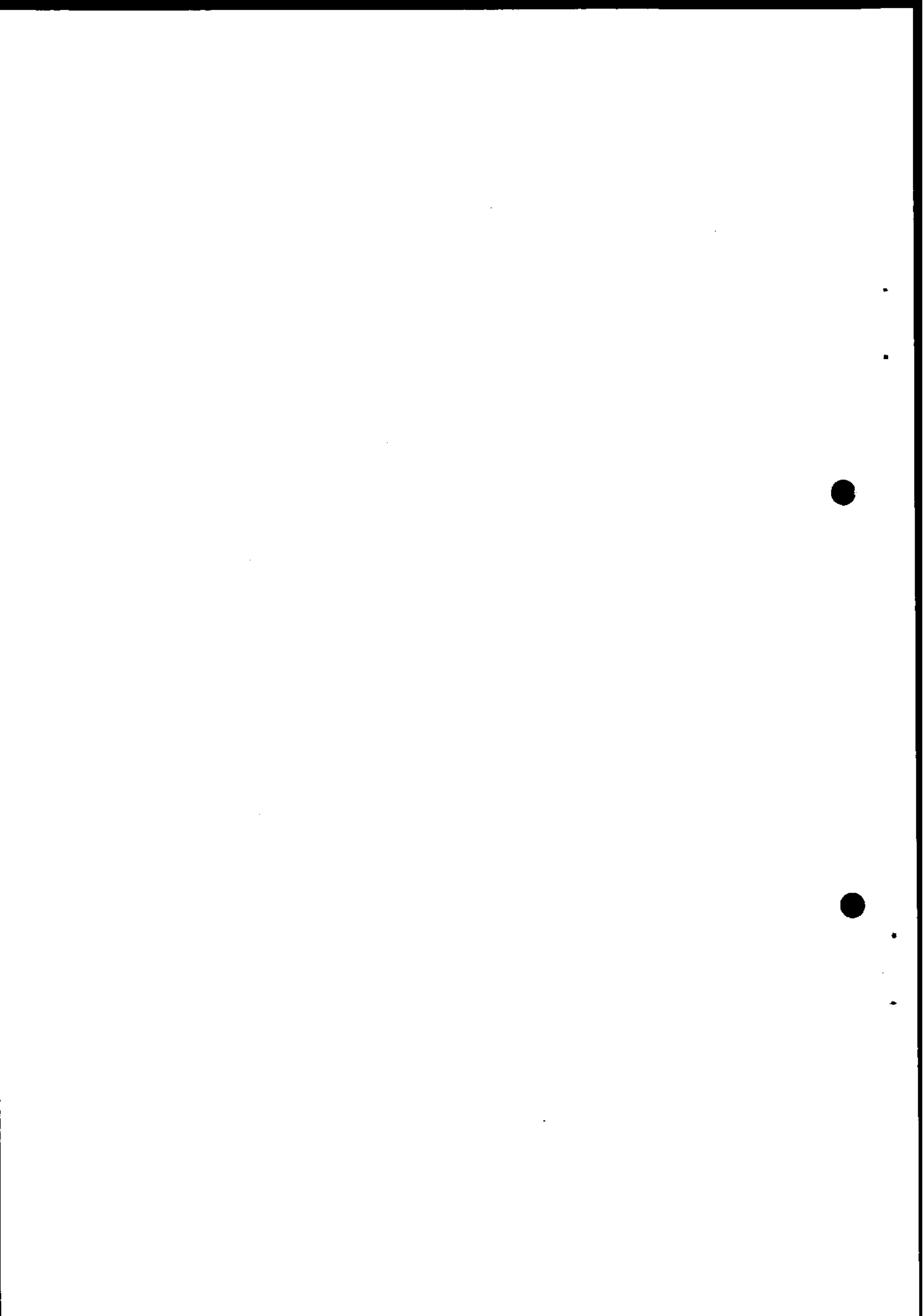
〔14〕 MOSES, J.: Toward a General Theory of Special Functions, C. ACM vol 15, 1972, pp550−554.

〔15〕 JOHNSON, S.C.: On the Problem of Recognizing Zero, J. ACM vol 18, 1971, pp559−565.

〔16〕 SAMMET, J.E.: Survey of Formula Manipulation, C. ACM vol 9, 1966, pp555−569.

〔17〕 WANG, P.S.: The Undecidability of the Existense of Zeros of Real Elementary Functions, J. ACM vol 21, 1974, pp586−589.

第7章　関数型言語



7. 関 数 型 言 語

7.1 はじめに

最近，従来型の汎用プログラミング言語（手続き型言語）に対比されるものとして，関数型言語（Functional Language）あるいは関数的言語と呼ばれる言語モデルや数学的体系が注目を集めている。関数型というものを特徴づける性質として，非手続き的（nonprocedural）あるいは副作用のない点（free from side effect）がまず挙げられる。FortranやAlgolに代表される従来型の汎用計算機言語が，ノイマン型計算機の命令に対応したアセンブラ表記によるステップ・バイ・ステップによる表現を高級化しているに過ぎないのに対し，関数型言語はλ計算〔1〕（λ－calculus）や組合せ論理〔2〕（Combinatory Logic）あるいは再帰関数論〔3〕などといった数学的体系に基いて構成されている点に特徴がある。関数型言語の実行は本質的にはノイマン計算機において見られるような状態遷移という概念を持たずその代わりにプログラム（あるいは表現）の形式を変形（通常reduction〔4〕と呼ばれている）することによって他の表現形式が生成され，この変形作用自体が計算に対応するという形態をとる。もちろん同様の働きをノイマン型計算機上でシミュレートすることもでき，また実際それに近いシステムも存在するが，関数型言語に内在する種々の性質はそのような従来の計算機構造に捉われない形態の新しい計算機アーキテクチャに適応できる可能性を持っている。それは1つには関数型言語が計算のメカニズムとして抽象のレベルが高いという点であり，これによって関数型言語で記述されたプログラムの数学的処理が容易になることが挙げられる。また別の性質としては関数型言語においては制御構造が前面に押し出されず記述されるため次に述べる並列処理やデータフローといった概念との整合性が良い点も指摘できる。

さて次に挙げる2つの性質は今日までハードウェアならびにソフトウェア両者の立場において追求されてきた大きな課題である。

(1) ハードウェア実行上の並列性

(2) ソフトウェアの検証性

従来，ハードとソフトはほぼ完全に分離された形態として，しかもそのインタフェースを暗黙の内に了解されたものとして保ってきた。これはそれぞれがそれぞれの立場で独立に発展し得たという意味において望ましい点もあったかも知れないが，逆に何らかの制約条件を互いに課すことによって，ある一定の枠内を打ち破ることができなかったと考えることもできる。近年になってデータフローモデルやBackusの提唱する新しい言語形態〔5〕など従来からのハード・ソフトの境界を越えた新たな考え方が登場した。これらはハードウェアの並列処理やソフトウェアの検証性といった問題を従来からのハード及びソフトの概念に捉われずに展開している点に特色があり，関数的な体系をその考え方の基礎としている点は注目される。本章では以上の点を踏まえながら，関数型言語の持つ特徴的な形態や性質，また個々のモデルによってどのような差違があるか，さらにこれらの言語を計算機上で実行するために必要とされる条件やその形態ならびにその問題点は何かなどについて考察する。

7.2 関数型言語の一般的特徴

関数型言語を広い意味で捉えると，代入(assignment)文やGO TO文を持たない非手続き的言語と同意であると考えられる。Landin〔6〕(1966)は関数型あるいは非手続き的よりも指示的(denotative)という表現の方が適切であると主張している。指示的とは命令的(imperative)の逆の意味であり，彼の定義によれば，関数型プログラムは「ある事物を他の事物によって記述する」表現法であり，あるモデルを形式化した場合における関数的な概念の役割は他の概念を記述するための標準形を与えることである。したがってこのような考え方に基づけば，述語論理やプロダクションシステム，ACTOR理論といったものもこの中に含まれてくる。しかしこの考え方はあまりにも一般化し過ぎており多岐に渡りすぎるので，本章では関数型言語の定義をλ計算や

それに準じる数学的関数形式に基づいて構成された言語に制限することとする。

関数型言語の一般的性質はその汎用モデルであるλ計算やこれをより一般化した組合せ論理を解析することによって明らかとなるがここではその中のいくつかを箇条書きすることとする。

○副作用のない点

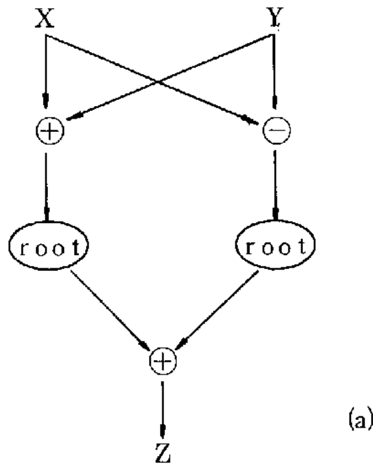
数学的な意味において，関数 y = f (x)は入力または引数 x と出力 y との関数あるいは写像を f という関数によって表現したものであり，f が定まっていれば出力 y は入力 x にのみ依存する。このような性質は関数の結合形にも適用される。したがって関数の値が定まらない場合を除けば，関数の演算過程は各関数の入出力関係（引数とその値）の連続形態として捉えることができ，そのような意味において状態という副作用が演算に影響することがない。

○並列性

λ計算においては関数の引数をどのような順序で評価してもその値は一致するので，各引数を並列的に評価してもよい。もちろんこの場合に各引数の評価において副作用がないことが前提になっていることは言うまでもない。この性質についてはλ計算の項でさらに詳しく述べる。

○データ駆動やデマンド駆動といった従来のシーケンシャルな制御構造を持たぬ新しいプログラム図式と結合する可能性を持つ。

データ駆動型の基本的な考え方は，あるオペレーションに対してそのオペランドの値が全て定まっているならば直ちにそのオペレーションの実行を可能とするようなモデルであるが，オペレーションを関数としてオペランドをその関数に対する引数値とみなせば，これはそのまま関数型言語の計算実行モデルに対応する。たとえば図 7 － 1 (a)に示すデータフロー図式は図 7 － 1 (b)の関数式に対応することが知れる。



(a)

$$Z = (+\ (\mathrm{root}\ (+\ X\ Y))(\mathrm{root}\ (+\ X\ Y)))$$

(b)

図７－１

○プログラム検証の容易さ

λ計算における数学的諸性質や組合せ論理と命題論理の関係を利用したり，関数型プログラムにおけるFixed point 理論（Scott）などを導入することにより，関数型言語プログラムの性質を数学的手段により把握することが関数的でないプログラムに比べ容易であることが知られている。プログラムの検証の方法はどのような関数型プログラム言語であるかによってさまざまであるが，Backusが指摘しているように関数型言語は，その検証を初歩的な代数を用いて行えるように言語を設定できる可能性を備えている。

7.3　λ計算〔1〕

λ計算を1つの計算モデルとして見た場合，チューリングマシンと本質的に等価なモデルを別の表現形式で表わしたに過ぎないと考えることもできるが，この表現形式の差異が各モデル上に構築された言語やシステムに対して少なからず反映することとなる。たとえばλ計算のモデルはチューリングマシンと異

り，状態という概念を持たないので，これに基く関数型言語には前節で述べたような副作用のない性質が生じる。

λ計算は純粋に関数を評価するための言語形式を与えるものであり，その形式は少数のプリミティブによって構成される。またこれを言語としてみた場合，その実行（インタプリート）の概念が言語内に埋め込まれており，このような意味においてこれを完全言語（complete language）（Backus 73）〔4〕と呼ぶこともできる。λ計算における実行シーケンスは各関数の持つ束縛変数に対して値を置き換えることに対応する。

7.3.1　λ計算の表現形式

λ計算の表現式（expression）を構成するための基本エレメントは変数でありこれを小文字で表わす。変数の集合をVで表わすこととする。変数の表現領域（domain）はすべての関数であり，以下に述べる表現形式が関数を表わす。λ計算で用いられるシンボルはVの集合とλ，(,)であり次の規則によってλ表現と呼ばれる関数が構成される。

1. 変数はそれ自体λ表現である。
2. もしMがλ表現で，xが変数ならばλxMはλ表現である。λxは束縛変数部と呼ばれ，Mはbodyと呼ばれる。
3. もしFとAが共にλ表現ならば（F　A）もλ表現である。Fはオペレータ部，Aはオペランド部と呼ばれる。

λ表現の例としては，x，λxx，λx（y（xy）），λxλyλz（xλyλxx）などが挙げられる。

7.3.2　λ計算の規則

λ表現における計算過程はあるλ表現が他のλ表現に変換（reduction）されることによって遂行される。ここであるλ表現M内の変数xをすべてあるアーギュメントAで置き換えた結果を$S^{x}_{A}M$と表わすとすると，λ計算の計算規

則は次のようになる。

1. （λxMA）はS^x_AMによって置き変えることができる。ここにおいてM内にはxをλ結合した形式が現われてはならない。またA内にMでλ結合されている変数が自由変数として現われてはならない。もしそのような変数が存在すれば次の規則によってこれを取り除く。
2. xがλ表現Mにおけるλ結合の変数であった場合，M内にxなる自由変数が同時に存在せず，かつM内にyという変数が現われなければ，MをS^x_yMで置き換えることができる。

次に上に述べた規則によってλ計算を行った例をいくつか示す。

$$(\lambda x(xy)\lambda zz)\xrightarrow[S^{x}_{zz}(xy)]{①}(\lambda zzy)\xrightarrow[S^{z}_{z}y]{①}y$$

$$(x(\lambda xx\lambda x(xy)))\xrightarrow[S^{x}_{x(xy)}x]{①}(x\lambda x(xy))$$

$$(\lambda x(x\lambda x(xy))z\lambda yy)\xrightarrow[S^{x}_{u}\lambda x(xy)]{②}$$

$$(\lambda x(x\lambda u(uy))z\lambda yy)\xrightarrow[S^{y}_{v}y]{②}$$

$$(\lambda x(x\lambda u(uy))z\lambda vv)\xrightarrow[S^{x}_{z\lambda vv}(x\lambda u(uy))]{①}$$

$$(z\lambda vv\lambda u(uy))$$

7.3.3 Church－Rosserの定理と並列性

前節で示したようにλ表現を計算ルールに従って変換してゆくと，それ以上変換できない表現形式となる。これがλ表現の評価値であるが，次の定理はあるλ表現を計算する場合の順序とその値の関係を明らかにする。

〔Church－Roserの定理〕

もしあるλ表現が別々のシーケンスで変換されたとしても，両シーケンスと

もその値が定まるならば，その値は同一のクラスに属する。すなわちλ式として同じ値になる。

あるシーケンスでは値を持つが，別のシーケンスでは計算が停止しないという場合もある。たとえば

（F　A）=（λxλyy（λx（x　x）λx（x　x）））

の計算において，M=λyyとしA=（λx（xx）λx（xx））として$S_{\lambda}^{x}M$で置き換えれば値は定まりλyyとなるが，一方最初に（λx（xx）λx（xx））を変換しようとしてもこの変換は停止しない。このようなことから停止しない変換シーケンスを含んだλ表現の取り扱いはやっかいであるが，次の定理による順序によって変換を行えば，1つでも停止するシーケンスを持ったλ表現は必ず値を持つ。このようなシーケンスのことをNormal Orderと呼んでいる。

〔定　理〕

λ表現の評価を左端のオペレータ・オペランド結合によって評価し続ければ，そのλ表現が定義され価を持つ時，またその時のみ停止する。

以上のことを踏まえて，λ計算やそれに基いた関数型言語の評価における並列実行の可能性について考察する。

並列実行の条件として次のような点が指摘されている。「あるプログラムの並列実行のためには，そのプログラムを表現している言語自体の持つある種のセマンティックスによってプログラム本体の並列解釈がなされるようにすることが重要である（Friedman 76）〔7〕。すなわちコンパイラやインタプリタによってあるプログラムの並列解釈を行うためには，そのプログラムあるいはそれを記述するための言語構造が単純で並列処理に適したものとなっていることが必要である。これには言語の構造として破壊的な代入文を持たないことや副作用のないことなどが含まれる。データフロー計算機用の言語として開発されたIDやVALなどもやはりこのような性質を備えている。

λ計算や関数型言語の評価における並列処理の可能性は前に述べたChurch－Rosserの定理により引数の評価を並列的に実行できる点にある。λ計算に

おいて複数（たとえばn個）の引数を持つ関数はλxMで表されるλ表現のbody M自体が（n－1）個の引数を持つ関数であると見なせばよい。この場合，すべてのλ変換の値が定まるならば定理によりどのような順序で変換を行っても全λ表現の値は一致するから，各引数を同時に評価したとしてもこのことは成立する。

たとえばf（s,t,u,v）=（s+t）+（u+v）をs=1，t=2，u=3，v=4に対して評価する場合，これをλ表現で表わせば

（（（（λsλtλuλv（（sum（（sum S）t））

（（sum u）v）））1）2）3）4）

となり，これを評価してゆくと

（（sum（（sum 1）2））（（sum 3）4））

なるλ表現が得られる。ここにおいてsumは和を計算するための関数であり（（sum 1）2）と（（sum 3）4）の評価は同時に実行することが可能である。

しかしながらLISPに見られるように，関数型言語に基いていてもシーケンシャルな処理を前提としたセマンティックスを持った言語においては上述した点は必ずしもあてはまらない。この点については次節以降でさらに述べることとする。

7.4 関数型言語の評価構造

関数型言語の性格を特徴付けるものとして

(1) 言語としてのシンタックス

(2) 言語を表現するためのデータ構造

(3) セマンティックスすなわち関数を評価する構造

の3つが重要な要素であると考えられる。(1)に関しては，関数型言語の本質的にapplicativeな性質，すなわちオペランドに対してあるオペレーション（または関数）をほどこすという操作を簡潔に表現するために何らかの識別子

を設ける必要があり，かっこやコロンなどといった特殊記号によってこれを表わすことが一般的である。

(2)のデータ構造は，関数型言語のインプリメントや効率という面を考慮した場合に大きな影響を与える要素である。たとえばＬＩＳＰ〔8〕はリストというデータ構造と関数型の制御構造を結びつけた言語として捉えることができ，リスト構造による柔軟なデータ表現によって関数型言語の利点を生かすことができた一例である。

(3)のプログラムにおけるセマンティックス構造は，ある言語で記述されたプログラムの振舞いを明らかにする理論とみなすことができ，この場合における言語のインタプリンタはその言語のセマンティックスに対する１つのモデルとなる。関数型言語のセマンティックスの基礎は数学的な関数の理論であり，ある表現（または式，expression）を関数や定数などをも含んだ抽象的なセマンティックスの値に対応付ける（写像する）ものであると考えることができる（Ward 74）〔9〕。したがって表現形式上同一の関数を表わしているものであっても，これを解釈する評価構造が異なれば，そのセマンティックスは当然異るものとなり，評価構造によって関数の表現能力（表現可能領域）に差異が生じる。Ward は関数の評価構造をモデル化し，個々のモデルの能力を考察した（Ward 74）〔9〕。これは関数型言語の記述能力とインプリメント方式との関連を考えるうえで大いに参考になる。以下では彼のモデルを引用しながら(3)について考察することとする。

7.4.1　再帰関数とスタツクモデル

再帰的関数（recursive function）とはある関数を定義する場合にその関数自身を直接または間接に参照することを許すものである。これは広い意味でくり返し演算モデルの拡張となっており，より大きな演算能力を持つ。再帰関数は広い意味ではλ計算とほぼ同じ意味で使われるが，ここでは後述するような狭い意味でのモデルとして捉える。

λ計算の実行規則で述べたように，関数型言語の計算は本質的には変数に対するアーギュメントの置き換えによる。これを行うのに，関数表現に対して直接置き換えを行い別の表現を生成する方法（いわゆる reduction によるもの）と，置き換えの環境を別に用意しておきこれを参照しながら計算を進めてゆく方法（evaluation）とに分類できよう。前者の例としてはλ計算のメカニズムやBerklingのマシン〔10〕〔11〕が挙げられる。また後者にはLISP のインタプリタ，SECDマシン〔13〕などがある。

evaluationによる計算における環境とは結合リスト（binding list）と呼ばれるordered pair（X, V）の列（$(X_1, V_1)(X_2, V_2)\cdots$）で表わすことができる。ここにおいてXはλ変数の名前であり，Vはその値である。本節で述べる狭い意味での再帰関数の評価モデルはこの環境構造を後入先出(last in first out)のスタック上に表現したものである。すなわちλ表現を評価する際に新たな結合対をスタック内に押し下げ，その評価が終了した時点でスタックをポップすることにより以前の環境に戻す操作を行う。このモデルにおける関数インタプリンタにおいては，あるλ変数Xの値はスタック上にストアされたordered pair をスタックの上部からサーチし，その最初の要素である名前がXと一致するものを探すことによって得ることができる。

本モデルによって関数型言語を形成した場合，あるデータ構造とこれに対するプリミティブな関数（機能）を規定すれば，本モデルのインタプリンタはこのデータ構造上の1階の再帰関数を評価することができる。ここでの1階の関数とは定められたデータ構造上によって表現し得る値のみを関数の引数や関数の値とするような関数のことである。たとえばデータ構造として数値の集合を与え，数値演算の基本オペレーションを付加すれば，このモデルに基いた関数集合は数値演算上完全なもの，すなわちどのような数値演算を行う関数をも表現し得ることが言える。

7.4.2 Funarg問題と木構造モデル

前節でのスタックモデルでは関数やλ表現自体を引数や値とするような関数（2階の関数）を表現することはできない。これはFunarg問題と呼ばれており，次のような例で示すことができる。

f〔x〕を次のように定義する。

f〔x〕＝次のような関数gによって定義される。

関数　g〔y〕＝x＋y

上の関数をスタックモデルで評価しようとしても，自由変数を含んだλ表現においてはその評価すべき環境が，ある関数が適用された後の変数の結合環境のみに依存してしまい，その関数を適用した時点の環境が保存されないので評価不可能となる。すなわちx＝3としてf〔3〕を評価した場合に，得られる値はg〔y〕＝x＋y という関数であり，ここにおいてxはgという関数においては自由変数である。スタックモデルではここでgを評価しようとしてもxの結合環境はf〔x〕を評価する以前の状態に戻されており，x＝3とはならない。

よりわかりやすくするためにLISPで記述した例を示す。

```
(LAMBDA (X)
        (QUOTE (LAMBDA (Y)(PLUS X Y))))
```

上のような関数Fが定義されていた時，変数Xの値が現在 10 であったとすると

(APPLY (F 3) 5) を実行しても値は8にはならずX＋Y＝10＋5＝15となる。

この問題を解決するためには別のモデルを導入することが要求される。1つの方法は環境を表現する手段としてスタックではなく木構造を用いるものであり，LISP1.5〔8〕ではこの方法が採用された。別の方法としては，結合環境を保存する手段として静的結合方式を用いることにより動的結合方式におけるFunarg問題自体を生じさせないようにすることである。これはSCHEME〔13〕〔14〕〔15〕で採用された。

前者の方法は，関数を表現するためにλ表現とその表現が評価されるべき環境の両方の情報を含んだclosure と呼ばれるものを用いるものである。これによってある関数から制御が戻ってきても，その関数評価時における環境を保於しておくことができる。前に述べたＬＩＳＰの例では関数Ｆを

```
(LAMBDA (X)
        (FUNCTION (LAMBDA (Y) (PLUS X Y))))
```

と表わすことによってこれを実現することができる。（F　3）を評価した時の値はこの場合

```
(FUNARG (LAMBDA (Y) (PLUS X Y))
        ((X . 3)))
```

となりこれを5に対してapply すれば，X＋Y＝3＋5＝8が得られる。

7.4.3　Lazy evaluation 方式

木構造を用いた前節の方式ではfunarg問題を解決することができたが，λ計算におけるnormal order による評価モデルとまだ一致しない面がある。それは関数に対する引数が必ず1度評価されてしまうという点である。一方λ計算においては引数の評価はそれが要求された時に限って行われる。したがって木構造モデルにおいては，ある引数の評価値が定まらない（無限評価ループに入る）ことによって全計算の値が定まらない場合が生じる。Reynolds[16]はこのような可能性のある関数をserious 関数，常に計算が停止するものをtrivial 関数と名付けている。たとえば（λxλyλz　x）で表わされるような，3つの引数のうちの最初の引数の値をその関数の値とするものを考えた場合，これをλ計算におけるnormal order で評価すれば，第2第3の引数の値が定まらなくてもその値は定まるが，これをＬＩＳＰ関数（LAMBDA （X Y Z） X）と見なせば，この値を求める際に3引数の値が定まらなければこの関数自体の値も定まらないことになる。

これを解決するには何らかの手段でλ計算におけるnormal order の評価

方法をシミュレートすることが要求される。ALGOLに導入されたcall by nameによる値参照はこのことを局所的に行ったものである。前節の木構造モデルの環境を保存しつつ，その上に別の評価構造を付加するモデルはlazy evaluation〔17〕方式と呼ばれている。

LISPにlazy evaluation 方式を導入した例を考えてみる。データ構造を作成する基本関数cons の実行を真にそのデータを参照する時まで遅らせる(suspend)ことによりこの方式はsuspended cons〔18〕と呼ばれている。たとえば

(car (cons (f x) (cons (g y) (h z))))

を評価する場合，初めのcons を評価した時にその引数は評価されずその時の関数の形体と環境を持ったsuspensionなるものに変換される。そしてcarを評価した時点で始めてそのsuspensionの評価が実行されるが，carを求める際には(cons (g y) (h z))を評価する必要がないので，このsuspension は永久に実行されることがない。

7.4.4 Continuation

λ計算的な関数型言語の機構に対して拡張をほどこしある種の制御メカニズムを導入することによって表現能力を高めることができる。これは続き関数(continuation function)と呼ばれており，ある関数 f_{old} に対して，continuation なる余分の引数がつけ加えられ新たな関数 f_{new} となる。f_{new} の値は fold の値にcontinuationなる関数を適用したものである。すなわち

$$f_{new}\ (x_1, \cdots\cdots x_n,\ c) = (\ f_{old}\ (x_1, \cdots\cdots,\ x_n\ ))$$

となる。ここにおける計算メカニズムは関数がリターンしてから次の関数を適用するというのではなく，次に実行すべき関数をcontinuation という引数に埋め込んでこれを他の関数に手渡すという形体をとっている所に特徴がある。

```
(DEFINE FACT
     (LAMBDA (N C)
          (IF (= N 0) (C 1)
               (FACT(- N 1) (LAMBDA (A)(C(* N A)))))))
```

(a)

```
(entering 1 FACT)(3 C1)                          C1=(LAMBDA (A) A)
(entering 2 FACT)(2 C2)   (leaving 0 FACT)  C2=(LAMBDA (A)
                                                (C (* N A)))
(enteriug 3 FACT)(1 C3)   (leaving 1 FACT)      (C=C1, N=3)
(entering 4 FACT)(0 C4)   (leaving 2 FACT)  C3=(LAMBDA (A)
                                                (C (* N A)))
(entering 1 C4)   (1)     (leaving 3 FACT)      (C=C2, N=2)
(entering 1 C3)   (1)     (leaving 0 C4)    C4=(LAMBDA (A)
                                                (C (* N A)))
(entering 1 C2)   (2)     (leaving 0 C3)        (C=C3, N=1)
(entering 1 C1)   (6)     (leaving 0 C2)
 (leaving 0 C1)
```

(b)

図７－２

たとえばSCHEME〔13〕〔14〕〔15〕においてcontinuationを用いて階乗を計算するプログラムは図７－２(a)のように表わされるが，この実行メカニズムとしてtail transfer（他の関数がある関数の値をその引数として用いない場合には関数呼び出しによってその戻りの処理も同時に行われるとする。）の機構を前提とするとこの実行シーケンスは(b)のようになる。これを見てわかるように，continuation関数の計算メカニズムはcontinuationと呼ばれる関数とそれが評価されるべき環境が次々に手渡されることによって計算が進行するものと捉えることもでき，これをさらに拡張してメッセージの授受による計算モデルとの対応をとることも容易であろう。

計算のメカニズムにおいては関数としてではなく制御（すなわち評価の順序）

のみが要求される場合がある。もちろんダミーの引数を評価するという手続きにより同様の制御をシミュレートすることはできるが，continuation を関数型言語に取り入れることにより，より明白な形でこれを表現することが可能となる。

7.5　関数型言語の形体

関数型言語を設定する際において考慮の対象になると思えるいくつかの点を列挙してみる。

○名前付け

関数の定義内に束縛された引数に名前付けを行うものと組合せ子（combinator）や演算子などを用いることにより引数の名前を取り去ったものという2種類の形体がある。前者は主としてλ計算に基くものであり，ＬＩＳＰやSCHEMEそれにSECDマシン用言語などにみられ，後者はAPL（APLは完全な関数型言語とは言い難いが）やBackusのFP などに見られる。簡単化された表現とその数学的処理のし易さという点からは後者の方がすぐれていると考えられるが，言語としての読み易さや書き易さという点では前者にやや利点があるであろう。SASL〔19〕言語において示されたように引数の名前付けを持って書かれたプログラムをコンパイラによって引数のない形式に変換し，これを実行時に処理するという方式も考えられる。

○合成関数

従来からの汎用計算機言語においては，サブルーチンやプロシージャの定義をコールするという形によってある定義体における階層構造が形成されていた。関数型言語ではこの点はごく自然な形で記述される。通常はＬＩＳＰにみられるように，ある関数の引数に他の関数の評価した値を与えるという記述のくり返しとなる。ＦＰでは関数合成のためのいくつかの形式を用意し，これによって新たな関数が合成されたことを明示する。さらに進んでやはりBackus のＦＦＰにおいては，関数を合成するための形式自体も定義することが可能とな

っている。これは一見記述能力を増大しているように見られるが，同様のことは生成された形式自体を評価することのできるシステムでは可能なのである。合成の形式を定める利点はむしろ生成される関数の能力や形態に制限を加えることで，検証のし易さを増大させようとするものである。

合成関数の評価を考えた場合，関数という閉じた世界においては主たる関数とその構成要素となっている別の関数の間の橋渡しをするものが必要である。この内容は個々の表現された関数の能力の大小に依存する。サブルーチンのようなものでは引数の値だけで十分であるが，より能力を増大すると環境をも含めたclosure と呼ばれる形態が必要とされる。こういったものはいわば関数を合成したことによるオーバーヘッドであり，並列処理システムとの対応を付ければこれは通信オーバーヘッドに対応する。したがって関数型言語において，関数の合成形式をどのようにするかという問題は特にインパクトの大きな問題であると考えられる。

○言語的枠組と可変部分

この問題はBackus の論文のなかでも論じられているが，通常には小さな枠組を持ちつつ可変部分に対して大きな拡張性を持っている言語が望ましい。したがってこれは前に述べた合成関数と大いに関連がある。しかしながら現実的なインプリメントや効率の面を考慮すればそのようにするのが困難である点は，ＬＩＳＰの例（基本的サブルーチン関数の増大）をみればわかる。ここでの課題は可変部分を言語構造の中にインプリメントも含めていかにすっきりと取り込んでいくかという問題である。

7.6　むすび

関数型言語の大まかな特徴やこれの評価モデルなどについて考察してきたが，関数型言語が将来における非ノイマン計算機と言われているものに対して適合する可能性を持っているものの，これを汎用計算機言語として定着させるには解決しなければならないいくつかの課題があることが判明した。特に関数型言

語の並列処理方式については重要な点でありながら，あまり研究が進んでおらずこれからの課題であろう。現在の段階においては，将来の計算機アーキテクチャに対して最もインパクトを与えるような機能をいかに単純化して表現するかという点がより重要である。そのためには，現存するプログラミング言語の得失を洗い出すと共に，関数プログラミング的な概念をいかにまとまった形として計算機構　にとり入れていくかという点をさらに調査することが必要である。

＜参考文献＞

〔1〕 Church, A.: The Calculi of Lambda-conversion, Ann. of Math. Studies 6, (1941)

〔2〕 Curry, H.B., and Feys, R.: Combinatory Logic, (1958)

〔3〕 Scott, D.: Lambda Calculas and Recursion Theory, Proc. of 3rd Scandinavian Logic Symposium, Apr. (1973)

〔4〕 Backus, J.: Programming Language Semantics and Closed Applicative Languages, ACM Symp. on Principles of Programming Languages, Oct. (1973)

〔5〕 Backus, J.: Can Programming Be Liberated from the Von Heuman Style ?, CACM, Vol. 21, №. 8 (1978)

〔6〕 Landin, P, J.: The Next 700 Programming Languages, CACM, Vol.9, №3 (1966)

〔7〕 Friedman, D, P.: The Impact of Applicative Programming, Proc. of the 1976 International Conference on Parallel Processing.

〔8〕 MgCarthy, J. et al, : The LISP 1.5 Programming Manual, MIT Press (1965)

〔9〕 Ward, S, A.: Functional Domains of Applicative Languages, MAC TR-136, MIT, (1974)

〔10〕 Berkling, K, J.: Reduction Languages for Reduction Machines, Proc. of the Second Annual Symposium on Computer Architecture (1975)

〔11〕 Berkling, K, J.: Computer Architecture for Correct Programming, Proc. of the Second Annual Symposium cn Computer Architecture (1978)

〔12〕 Landin, P, J.: The mechanical evaluation of expressions, Computer J. (Jan. 1964)

〔13〕 Steel Jr, G, L. and G.J. Sussman: LAMBDA The Vltimate Imperative, MIT AI Memo №353, (March, 1976)

〔14〕 Steel Jr, G, L.: LAMBDA The Vtimate Declarative, MIT AI Memo 379 (Nov. 1976)

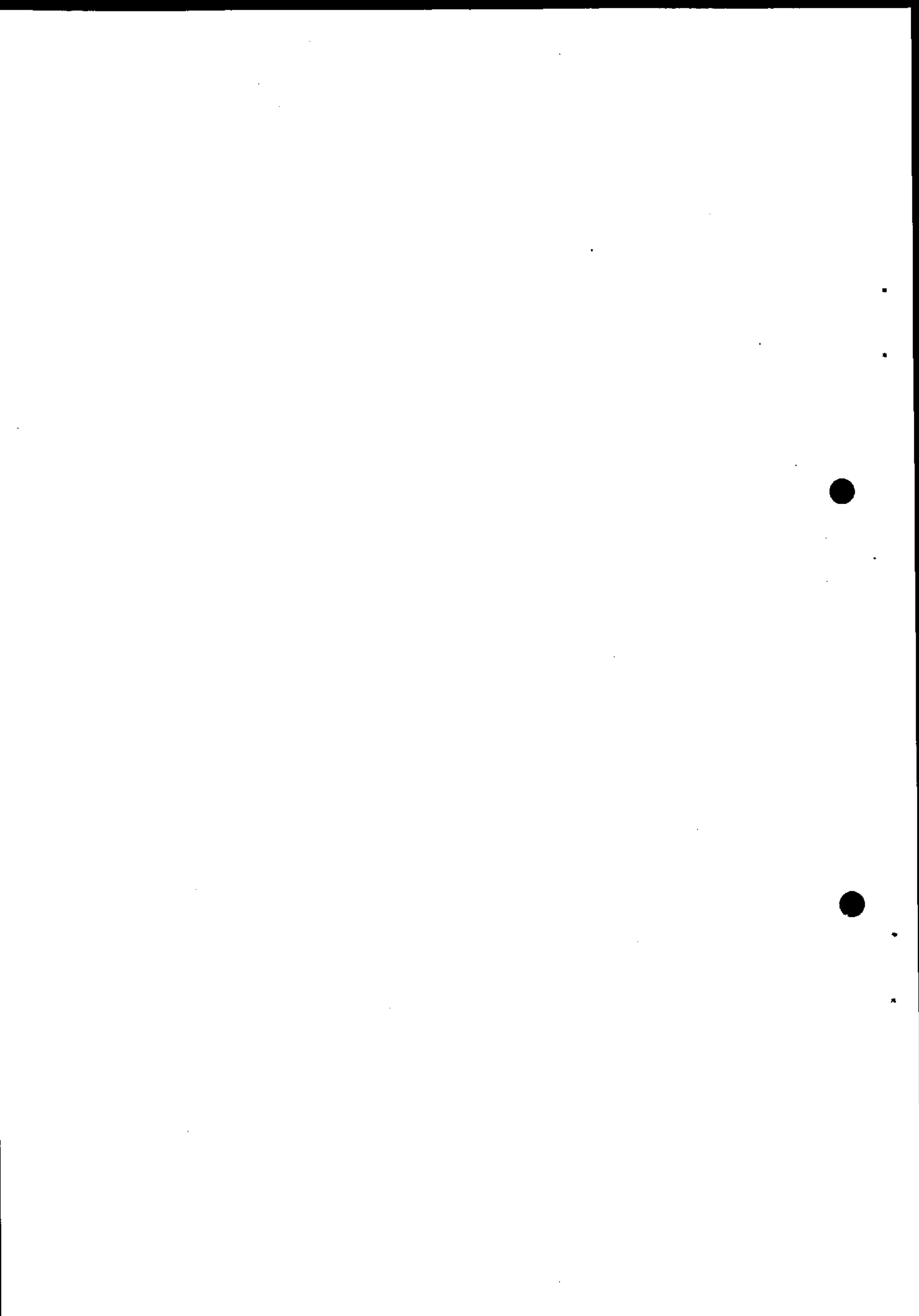
〔15〕 Steel Jr, G, L. and G.J. Sussman: The Revised Report on SCHEME, MIT AI Memo №452 (Jan. 1978)

〔16〕 Reynolds, J, C.: Definitional Interpreters for Higher-Order Programming Languages, Proc. 25th National ACM Conference, (Aug. 1972)

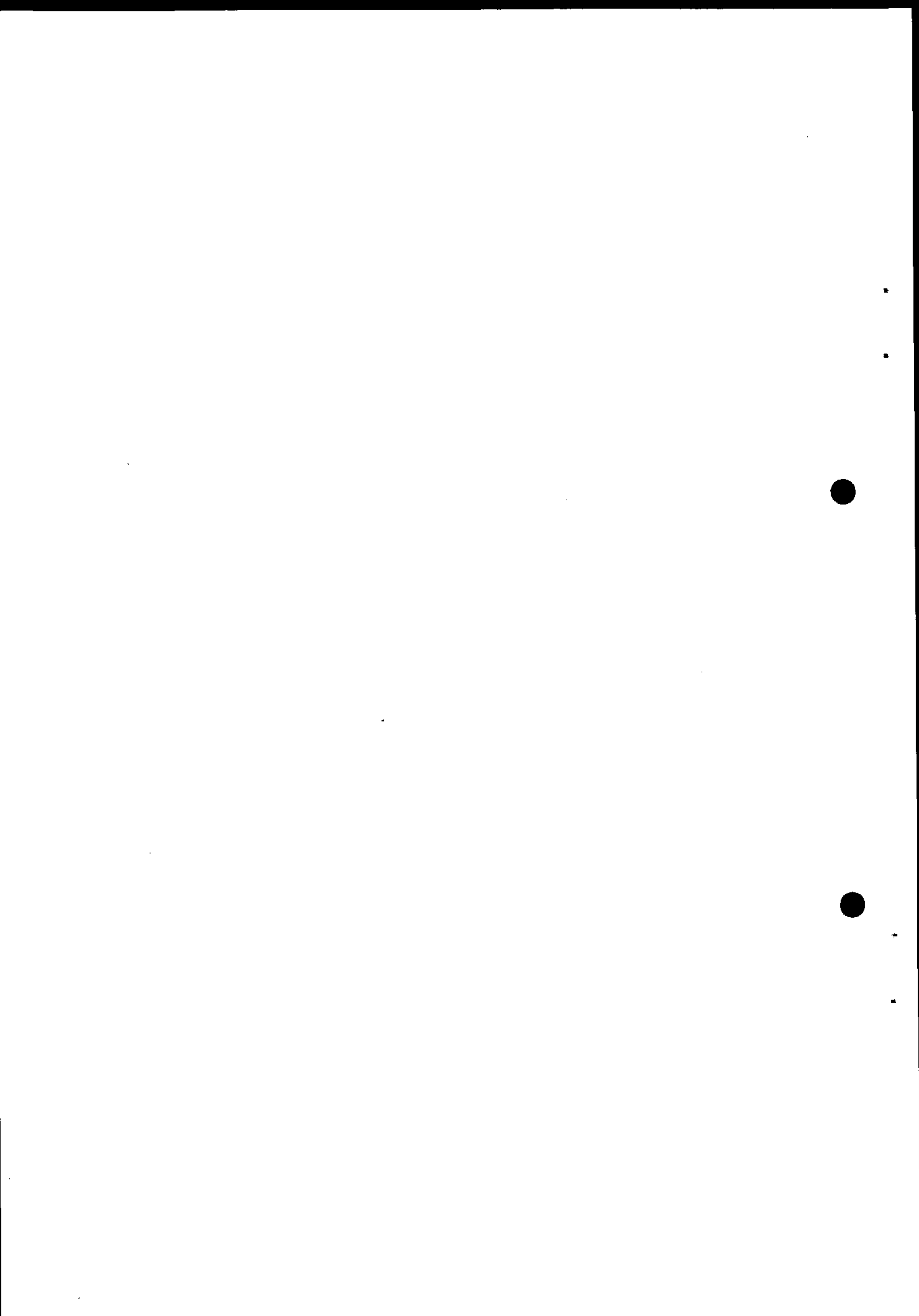
〔17〕 Henderson, P.: A Lazy Evaluator, 3rd ACM Symposium cn Principles of Programming Languages, (Jan. 1976)

〔18〕 Friedman, D.P. and D.S. Wise: CONS Should not Evaluate its Arguments, Automate, Languages and Programmjng, Edinburgh V. Press, Edinburgh, (1976)

〔19〕 Turner, D, A.: A New Implementation Technique for Applicative Languages, Software-Practice and Experience, Vol. 9 (1979)



第８章　駆動型プログラミング



8. 駆動型プログラミング

8.1 はじめに

最近，駆動型プログラミングや駆動型計算モデルが注目を集めている。これは，ますます巨大化，複雑化するソフトウェアを，いかに能率よく作成し，信頼度の高いものにするかという問題や，並列処理による高速化，超ＬＳＩ（ＶＬＳＩ）の利用を考えるとき，従来のフォンノイマン型計算モデルの持ついくつかの欠点が明らかになり，これに代るものが求められるようになってきたことによる。これら欠点のいくつかは，かって，少量の計算や記憶のためのハードウェアしか入手できず，アーキテクチャの簡単化が，最優先した時代に，実現上の利点となっていたものである。この簡単な構造が，高度な並列処理や，プログラムの整構造化を目ざす上で，障害となり，顕著な問題となってきたのは，ハードウェアの低価格化の達成が，従来の価値観を，転換させるに至ったからとも考えられよう。

駆動型プログラミングの基本となる考えは，フォンノイマン・モデルの逐次実行型の制御に代わり，データ，もしくは，メッセージによって，演算処理を起動しようとする考え方である。これは，データ駆動，または，データフロー・モデル，メッセージ駆動型モデルと呼ばれている。

このような計算モデルは，演算処理を行うオペレータの入力がそろい次第，その演算処理が駆動されることから，非同期的な並列実行を基本的に含んでおり，フォンノイマン型に比べ高水準のモデルとなっている。この特徴によって，従来のアレープロセッサ等に比べ，汎用性に富む並列計算機の実現に適した計算モデルとして期待されている。また，その構成にあたっては，高度のモジュール化が行えるため，演算処理モジュールや記憶モジュールなど同種のモジュールの多数のくり返し構成がとれ，ＶＬＳＩ化に適するものと考えられている。しかしながら，データ駆動に基く，計算機の実現のためには，未知の問題

が多く存在しており，いろいろな方面から，研究がすすめられている。

データ駆動型モデルと，それに基く，データフロー計算機の研究としては，まず，上で述べた，汎用性を備えた並列計算機を目ざす分野があり，その理論的研究は，１９７０年代の初めから行われていた(1)(2)(14)。その後，データフロー計算機が重要視されるようになった背景には，並列計算機を求める分野とは直接関係のない，ソフトウェア工学や，人工知能の分野から出された，計算モデルや，プログラミング方法論，プログラミング言語等の研究成果からの影響が大きい。

ソフトウェア工学の分野では，プログラムの作成，検証，変換，拡張，保守などを容易にし，その一部を自動化することを目ざしているが，近年，そのような目的に適合するようなプログラミング言語や，その数学的性質が徐々に明らかにされてきた(3)。抽象データ型やカプセル化によるモジュール化の手法，オブジェクトのモデル化のいろいろな手法，関数性と副作用などの関連性の明確化などは，このような研究の成果である。これらを用いた最近のプログラミングでは，データとそのオペレーションを一まとめとして，より抽象的なデータ型とし，より高レベルのプログラミングを行おうとするものが多い。この結果，プログラマは，計算機に依存するような細部の記述にわずらわされること無く，いろいろな問題の記述が，より抽象的なレベルで可能となる。このような，データを中心と考えるプログラミング手法は，従来のフォンノイマン・モデルよりもデータフロー・モデルによって，より自然に実現できると思われる点も多い。（このような考え方は，現在，まだ一般的ではない。）また，プログラムの正当性の証明などのためには，数学的な基礎を持つ関数型プログラミングや述語プログラミングが適しているが，これらの研究の成果は，フォンノイマン・モデルを支持しているとは言い難い。

人工知能研究の分野において計算機と密接に関連するテーマとしては，知識表現のための手法や記述のための言語の研究がある(10)(11)。ここでは，記述の対象となる知識の形式化に適したモデルが選ばれ，データフロー計算機と直接の

関連は無い。しかしながら，計算機上で，記述した知識をたくわえ，推論のために検索する等の操作を行う必要から，より具体的な，プログラミング言語や計算モデルが求められ，その結果として得られたものを見る時，オブジェクト指向型のプログラミングや，データ駆動型モデルに近い形式が見られる。このような問題は，将来の計算機の応用分野として，重要なものであり，将来の計算機が満たすべき要件を示すものとして，深く関係すると考えられる。

データ駆動，メッセージ駆動に基く計算モデルや，これをもとにしたプログラミングの研究は，新しい計算機システムや，そのアーキテクチャの望ましい姿を求めるための第一歩と考えられる。この研究は，多くの分野と，密接に関連しているが，ここでは，従来からの並列計算機を求める研究の流れ，ソフトウェア工学との関連性，人工知能における知識表現や言語との関連性について，述べる。

8.2　並列処理マシンを構築する動き

8.2.1　まえがき

8.1では，様々の分野からデータフロー計算機，あるいはデータフロー・モデルに対するアプローチがあることを述べた。本章では，それらのうちから並列処理マシンとしての側面を中心にとらえ，その動向を考察してみることにする。現在，アメリカ，そしてヨーロッパの，少なくとも十ケ所以上の研究所や大学においてデータフロー計算機の研究が行なわれており，試作機も何台か作られている。しかしながら，データフロー計算機の研究動向を考察するにあたり，それらにおいて提案されているアーキテクチャだけをサーベイして，それらからボトムアップ的にデータフロー計算機の全体像，そして研究動向をさぐるという方法は適切とはいえない。なぜならデータフロー計算機は従来のフォンノイマン・モデルとは根本的に異なった計算モデルに基づいているために，データフロー計算のモデル化，およびそれらをサポートできるような，新たなプロ

グラミング言語の設計を行なってからでないと，計算機アーキテクチャの実質的な検討にかかれないという面が多分にあるからである。研究の現状を見ても，実際的なアーキテクチャの研究よりも言語面の研究が先んじているといえる。そしてこの事実は，現段階におけるデータフロー計算機の研究の主体がソフトウェアサイドにあるべきであるという示唆とも受けとれる。そのような理由から，本章では子細な計算機アーキテクチャの提案にとらわれずに，計算モデルとしてのデータフロー計算機という観点に重きを置き，いわばトップダウン的に，並列処理マシンとしての要件，およびそれらに起因する問題点を考えることにする。まず8.2.2で，データフロー計算機の研究背景を述べ，さらに8.2.3では並列処理に焦点を絞り，その問題点を明らかにする。8.2.4ではデータフロー・モデルの性質がそれらの問題点に対してどのように有効であるかを述べる。最後に8.2.5で，データフロー計算機を構築する前に解決しなければならない事柄について考察する。

8.2.2 データフロー計算機の研究背景

データフロー計算機の研究動機は，過去三十数年間，計算機アーキテクチャ，プログラミング言語の基礎としてあったフォンノイマン・モデルから脱却することによって，現在よく論議されるソフトウェア危機，フォンノイマン型マシンの性能の限界といった，現在の計算機システムの抱える問題点を解決しようというものである。本節では，後者のフォンノイマン型マシンの性能の限界について考察を加えることにする。

高度の処理能力を要求される場合，現在における計算パワーの主力は，IBM 37Ø等の大型計算機，そしてSTAR−1ØØ，CRAY−1といったスーパーコンピュータにあるといえる。しかしこれらが将来の計算需要の増大に対処できるかどうかについては非観的な見通しが強い。大型機における，命令のフェッチと実行のオーバーラップ，命令先読みなどの手段は，あくまでも逐次制御を前提とした上でのアーキテクチャの改良であり，本質的に並列性が高く，分散

制御を必要とするような計算を十分にカバーすることはできない。またスーパーコンピュータにおいては，配列やベクトルについての処理は高速であるものの，スカラー演算が多くなると効率が低下するため，汎用性という点で難がある。そして大型計算機，およびスーパーコンピュータの双方に共通する最大の欠点として，これらがＥＣＬのような論理デバイスの高速性に依存しており，アーキテクチャからみて，ＶＬＳＩを有効に利用した，くり返し構造をとれないことがあげられる。つまり将来的展望においては，論理素子，記憶素子の高速化に頼る単一プロセッサの逐次処理よりも，プロセッサ・チップを多数結合して計算パワーを得るマルチプロセッサシステムの方が有望であるという見方が一般的になりつつある。しかしながら，これまでの計算機複合体の研究は決して十分な成功をおさめているとは言い難く，たとえ今ＶＬＳＩが実用化されたとしても，その大きな可能性をひき出すことはむずかしいと思われる。その理由として，多数のプロセッサを接続する場合の接続方法，資源配分方法を決める理論的基礎が不明確なこと，一般的な問題を並列処理が可能な形態に分割することのむずかしさ，並列処理を記述する使い易いプログラミング言語が準備されていないこと等があげられる。しかし根本的に見るならば，高度に非同期的で並列性の高い問題を，フォンノイマン・モデルの枠組の中で記述，解析をしようとするところにそもそも問題があるといえる。最近では，ハードウェア技術に問題があるというよりも，並列プログラムのモデル化の方法や，並列処理記述言語に対する考え方が，暗黙のうちにフォンノイマン・モデルに従ったものとなっており，これがプログラムの検証，並列プロセスの検出などのほか，ハードウェアの構成を考える上でも障害となっていると考えられるようになっている。

8.2.3　並列処理の問題点

前節では，従来の計算機複合体の実用化の障害の一つが，フォンノイマン・モデルの性質に由来していることを述べた。本節では，そういった観点から，

並列処理の問題点をより具体的に検討するために，「既存言語の並列性表現能力の欠如」，および「副作用の存在」の二点を取り上げ，それらの影響を考察することにする。

(1) 並列性表現能力の問題

並列処理記述言語の理想からすれば，並列性検出のための情報を書かず，単にアルゴリズムを記述するだけで，並列性，および実行順序の制約条件を表現できることが望ましい。しかし，フォンノイマン型言語，つまりメモリセルに対する逐次的な代入と参照を基本とする言語は，プログラマに対して並列性情報の明示を要求する。これによってプログラマはアルゴリズム以外にも注意を払うことを余儀なくされる。たとえばFORK文，JOIN文を使用する場合，もし共有変数に対して，読み出しの文と書き込みの文が誤まって並列に実行されると，結果は非確定性（indeterminacy）を呈することになり，デバックは困難になる。また他の例として，特別な並列処理機能を有する計算機システムの場合があり，それらにおいてはコンパイラの並列性検出を容易にするために個々の計算機アーキテクチャがプログラミング言語に反映されて，プログラマに対して問題解法の不自然な考え方を強制する。たとえばイリアックⅣのための専用言語GLYPNIRの場合，マシン内の最大64の並列性を生かすためにプログラマはアルゴリズムの選択，そしてデータの構成についても検討することを要求される。以上掲げたようなプログラマに対する負担は，メモリセルにたえず状態を保持し，それに対する内容の更新を根本の方式に据えたフォンノイマン・モデルの枠組の中では，いかに言語，アーキテクチャを工夫しても，程度の差こそあれ，本質的につ いてまわるものである。計算モデルに対する見直しをはかる以外に抜本的な対策は見出しにくいといえる。また仮りに，こういった負担がソフトウェア開発者の中で許容されたとしても，既存の言語によって表現できる微細な並列性には自ら限度がある。たとえばセマフォを用いる場合，プロセススケジューリングとwait, signal 操作のオーバーヘッドを考えると，とても文レ

ベルの並列性の記述には適用できない。またモニタの場合，共有変数（資源）に対するアクセスを排他制御にしているために，たとえ並列に実行可能なプロセスがあったとしても，それらの並列性を生かすことはできない。現在までのところ，文レベル以下，つまり算術式内の微細な並列性をも効率よく表現できるような言語をフォンノイマン型言語の範ちゅうの中に見出すことはむずかしい。

(2) 副作用（side effect）の存在

並列処理プログラムの正しい実行結果を保証するためには副作用のないことが第一の要件である(14)。一般に副作用という言葉について明確な定義はなされていないので，本稿では原則として次のような意味で用いる。すなわち，時間に依存した情報を保持する変数を二つ以上のプロセスが共有する場合，どちらか一つのプロセスの実行によって，その変数の状態が変化し，それによって他のプロセスに予め明記されていない効果が生じることを副作用と呼ぶことにする。典型的なケースとして，コールしたサブプログラムの中で，全域変数を更新する場合をあげる（図8－1）

```
procedure GETRS ( X,Y:real);(*RS is declared in
begin RS:=X*X+Y*Y              an outer block*)
end;
```

図8－1

もし全域変数ＲＳに対して更新を行なうプロセスが，このサブプログラムGETRSと同時に実行されたとすると，プロセス相互の時間関係によって，正当な結果が得られない場合が生ずる（非確定性）。つまり同一の初期条件において実行を再現したとしても同一の出力が保証できなくなる。このような副作用を除去する方策としては，まずcall by reference, call by name に代えてcall by value を採用すること，そして全域変数を廃止することが考えられる。またあらゆる変数に対して厳密な有効範囲規則(scope

rule）を設けて，影響の局所性（locality of effect）を保つことも重要である。しかしこれらの，フォンノイマン型言語の中での方策は言語に対する表面上の制約に留まっており，本質的には依然として副作用の可能性を含んでいる。

また，並列処理における副作用の存在は明らかに有害ではあるものの，副作用を完全に除去するかどうかについては，その是非を一面的には論じられないということをつけ加えておく。なぜならば，フォンノイマン・モデルは全面的にメモリセルの更新を利用しており，これも広義の副作用としてとらえることができるからである。つまりプログラマが意図的に副作用を起こすことによって計算を進行させているという見方が成り立つ20）。このため，副作用を完全に除去すると従来通りに計算を記述できるのかという疑問が生じ，それが同時に8.2.5で述べるデータフロー計算機に対する制約につながる。

8.2.4 データフロー・モデルの性質

前節で掲げた"並列性表現能力の欠如"，"副作用の存在"という二つの問題点に対して，データフロー・モデルは本質的，かつ原理的な解決を与えることができる。前者に対しては，逐次制御からデータ駆動型制御への転換によって，問題に潜む最大の並列性の表現を可能にしている。後者に対しては，フォンノイマン・モデルが言語面へのきびしい制約によって副作用を防止しようとするのに対し，データフロー・モデルはバリュー指向型のアプローチをとることによって，副作用の原理的除去を実現している。以下，データ駆動型制御，バリュー指向型の二点について詳述する。

(1) データ駆動型制御

データフロー・モデルの最も有用な性質は，命令の実行が逐次制御型によるのではなく，その命令の必要とするデータが整い次第実行が可能になるという「データ駆動型」の概念に基づいていることである。今，図8－2のようなプログラムがあるとする。もし何らかの並列処理機能があって，このプログ

グラムを最も速く実行しようとすると，その実行順序は，①文１②文２と文３の並列実行③文４と文５の並列実行④文６，のようになり，４ステップで終了する。この実行順序はオペランドデータの流れが実行の制御情報となる「データ駆動型」に従うものであり，このような制御方式をとることによって，問題固有の並列性を生かすことができる。同期をとるための特別なプログラム構造を必要とせずに，文レベル以下の，算術式内での微細な並列性をも「データの従属関係」から自然に表現することができる。

このようなデータ駆動型の実行形態を記述するためには，図８－３に示すようなデータフロー・グラフが用いられる。図8－3は，図８－２のプログラムに対応するものである。　丸印で囲まれたノードはオペレータ（演算処理を行なうもの）と呼ばれる。オペレータ間を結ぶアークはデータの流れを示すもので，これは同時にコントロールの推移も表現している。

```
1  P=X+Y
2  Q=P／Y
3  R=X*Y
4  S=R−Q
5  T=R*P
6  RE SUL T=S／T
```

図８－２

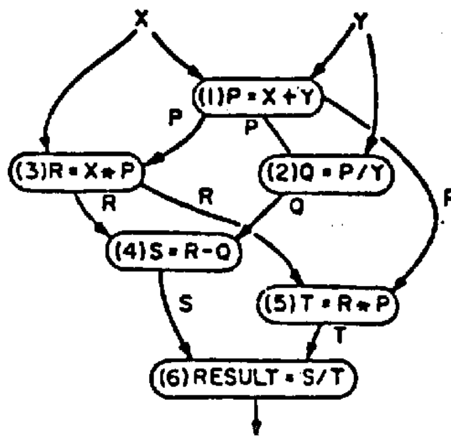


図８－３

これまでの説明から明らかなように，データフロー・モデルは各オペレータによる非同期的な分散制御を基本としている。このため，オペレータの機能を処理ユニットに対応させた並列処理マシンを構成しようという発想に自然に結びつく。

(2) バリュー指向型のアプローチによる副作用の除去

図8－2のプログラムは，「データの従属関係」をプログラムの表記から解釈することによって問題固有の並列性に応じた処理を行なうことができた。しかし今，B〔J〕:＝4；T:＝3＊B〔J〕；M:＝B〔K〕；というプログラムがあるとすると，コンパイル時にこれらが並列に処理できるかどうかを知ることはできない（J≒Kであれば，1番目と3番目の文は並列に実行できる）。この例は配列を含むプログラムの場合には並列性検出のための機構をプログラムの表記法に加える必要があることを単に意味しているだけかに見える。しかし，これは究極的にはメモリセルの概念の否定につながっている。つまり，配列に象徴されるような共有データをメモリに保持し，それに対する書き換え，読み出しのくり返しによって計算を行なっている限りは，必ずその実行順序に対して何らかの制約条件が必要となってくることを示唆している。このようなことから，データフロー・モデルではメモリセルに絶えずデータバリューを保持するのではなく，必要な時だけ（つまりオペレータの実行結果が生成され，それを次のオペレータに渡すとき），そのデータバリューを有効にするというアプローチをとっている。あるオペレータがオペランドを処理しようとする場合，フォンノイマン・モデルではアドレスとデータのバリューが不可分のものとして扱われているので，オペレータはアドレスを使って内容をアクセスし，更新する（このため副作用の危険性が潜在的に含まれる）が，データフロー・モデルでは，オペランドは直接バリューとしてオペレータに処理される。そのバリュー自体はオペレータによって消費されて論理的には消滅し，新しい処理結果との置き換えが生じる。このような方式をここではバリュー指向型のアプローチと呼ぶことにするが，オペランドの処理に

よる影響はこの置き換えだけである。このため，データフロー・モデルにおける実行環境は個々のオペレータ自体であるという，きわめてローカルなモジュラリティが実現され，副作用の根本的な除去が可能になる。このような方式をとることによって，結果的に，データの共有と更新，そしてメモリセルの概念（アドレス）がなくなる。したがって，配列やレコードのようなデータの集合体でさえも，データバリューの集合としてデータフロー・グラフ上を流れる。当然のこととして，たとえば配列XのK番目の要素を6に書き換えようとする，X〔K〕：＝6；といった文は無効になり，代わりにそれに最も近い操作として，K番目の要素が6であり，その他の要素はすべて配列Xに等しいような配列を新たに生成する操作が用意される。MITのデータフロー高級言語VAL 15)，16)では，この操作がX〔K：6〕によって表現される。

(1)　B：＝A〔J：S〕：
(2)　C：＝A〔K：T〕：
(3)　P：＝A〔L〕：
(4)　Q：＝B〔M〕：
(5)　R：＝C〔N〕

図8－4　VALのプログラム例

今，VALによって記された，配列の計算のプログラムがあるとする（図8－4参照）。フォンノイマン型言語では，本節の冒頭に掲げた例のような問題点が生じるが，バリュー指向型のアプローチに従う図8－4のプログラムにおいては，文1と文2と文3は添字が互いにどんな値をとろうと並列に実行することができる。文4と文5は，プログラムによって示されるデータの従属関係に従って，それぞれ文1，文2の終了後に実行することができる。以上述べたバリュー指向のアプローチは，他のapplicativeな言語にも通じる特質であり，フォンノイマン・モデルとはきわ立った対照をなしている。

8.2.5 データフロー・計算機に対する課題

データフロー計算機内で実現される高度の並列性は，副作用を完全に除去することによって達成しうるものである。そして副作用を除去するためにメモリセルの更新の概念をなくし，バリュー指向型の考え方を導入した。本節では，このような性質がデータフロー計算機に対してどのような課題をもたらすのかを考える。まず，副作用の除去をサポートするメモリシステムの実現形態について検討する。次にメモリセルの概念をなくすことによって，副次的に生ずる問題として，経過依存性（history sensitive）の計算の取り扱いについて考える。

(1) メモリシステム

データフロー・モデルはバリュー指向型のアプローチをとるために，あるデータバリューが演算処理されると，そのデータバリューは消滅して，結果のデータバリューが新たに生成される。配列のような集合体の場合も同じで，処理のたびごとに新しいデータバリューの集合が生成される。したがって，メモリシステムに対する処理要求が増加し，システム内のボトルネックになる可能性が生ずる。ところが，配列などの処理において生成される新しい集合に注目してみると，もとのデータバリューと全く異なるというよりも，その一部分だけが変更される場合が多い。そこで，変更されない部分はメモリ上で共有し，論理的には各データ集合ごとの独立性を保てる機能を採用すれば，メモリシステムの負担を軽減することができる。この方式を最初にとり入れたのはＭＩＴのDennisによるグラフ言語であり，append とselect アクタの導入によって，副作用のないデータ構造操作を実現している。図8－5における'app s' という命令は，入力データαの部分木ｓの内容をデータγで置き換えた，新たなデータβを出力として生成する。αもβも実際には部分木γにおいてデータεを共有しているにもかかわらず，αとβの論理的な独立性は完全に保たれている。このような機構を下位での実行形態として想定している高級言語には，VAL，カルフォルニア大学（Irvine）のID19)

がある。

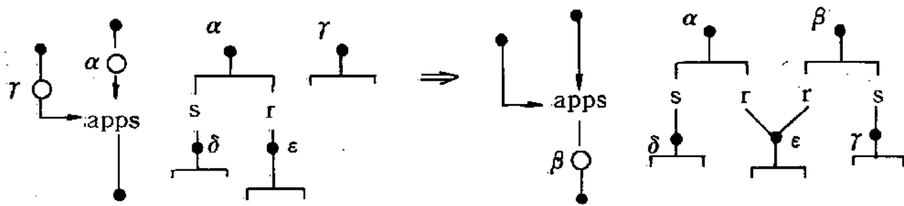


図８－５　アペンドアクタの原理

このような言語面におけるデータ構造操作をサポートするためのハードウェアとして，これまでに提案されているのは，ＭＩＴのAckerman によるストラクチャ・メモリ，及びその管理を司るストラクチャ・コントローラである。その原理を図８－６を用いて説明する。

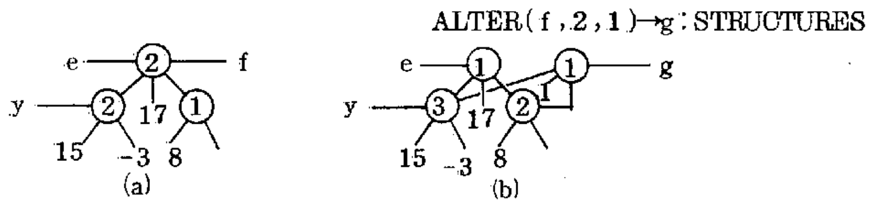


図８－６　ストラクチャコントローラの原理

配列やレコードは一般にツリー構造で表現することができる。図８－６(a)において，e, f, y というポインタは各々独立にツリー状データに参照を行なっているとする。ツリーのノード部分にある数はリファレンス・カウントと呼ばれ，そのノードを参照しているポインタの数を示している。さて今，ＡＬＴ－ＥＲ(ｆ，２，１)→ｇ：ｓｔｒｕｃｔｕｒｅという命令が来るとする。これはｆで参照されるトップ・ノードにおいて，左から２番目の部分木のデータである１７を１に置きかえて，それをｇというポインタでさせという命令である。これを行なうためには，単に１７を１にかえればいいかに見えるが，それを許してしまうと，何の変化も生じるはずのない，ポインタｅから見たデータ構造が

変化してしまう。つまり，e に対して副作用が発生する。このため図８－６(b)に示すように，新たにポインタｇのためのノードを作り，そこから，変更されない，もとのデータ部分に対してポインタをつなぐ。これに伴って，各ノードを指すポインタの数が変化するので，リファレンス・カウントの調整が行なわれ，たとえばポインタｙの指すノードのリファレンス・カウントは２から３になる。ストラクチャ・メモリとは，上記のような目的を持つリファレンス・カウントと実際の構造データを合わせて格納するものである。ストラクチャ・コントローラは，リファレンス・カウントを制御することによって，データフロー・プログラムの実行に伴って生成，消滅する構造データの管理を，副作用を生じさせずに実現するものである。しかしながら，これらはいわばマクロなレベルでの仕様を決めたにすぎず，実装の段階に至るまでには，メモリ技術の動向も含め，解決すべき問題も多い。どのような形態で実現されるにしても，データフロー計算機におけるメモリシステムは，従来のそれに比べて，複雑で処理量も多くなることはほぼ明らかである。しかし，それは副作用の除去によって得られる高度の並列性と引き換えになるのであるから，多少の負担は許容されるべきであろう。

(2) 経過依存性（history sensitive）

高度な並列性を達成するための副作用の除去は，同時にメモリセルによる情報の明示的（explicit）な保持の概念をも除去している。この結果，出力の決定に際し，過去の状態情報を必要とするような計算，つまりデータベースの更新に代表される経過依存性の計算に対して新たな方策が必要となる。また，つきつめて考えれば，I／Oも副作用の一種であり，メモリセルの状態保存を必要とするわけであるから，やはり何らかの手段が新たに講じられなくてはならない。この問題は単にデータフロー・モデルだけが抱えるものではなく，ラムダ計算の置き換えに基くシステム，たとえばBackus のＦＦＰにも見られるものである（ＡＳＴサブシステムの導入はそのためである）。これを解決するためにデータフロー・モデルにおいて提案されている

ものとしては，ＭＩＴのWengによるデータフロー言語TDFL17),Arvind によるＩＤにおいて導入されているストリームの概念がある。ストリームは配列に対する拡張概念であり，配列の各要素が一つずつ現われ，一要素ずつ処理されることを許すものである。例として，ＩＤのプログラムを図８－７に掲げる。

```
W -(initial v←v0;w←w0; t←t0
     for each status in V do
         new t ← status.time ;
         new v ← status.voltage;
         new w ← w+(v-w)*(1-e↑(-τ*(new t-t)))
     return all <time:new t, voltage:new w>)
```

図８－７　ＩＤによるプログラム

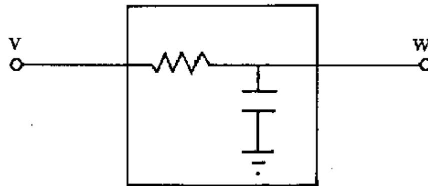


図８－８　式(1)に対するＲＣ回路

これは図８－８のＲＣ回路における入力，出力電圧の時間変化を計算するものである。ただし入力電圧ｖと出力電圧ｗは式(1)によって離散的に変化するものとする。

$W_{i+1} = W_i + (V_i - W_i)(1 - e^{-\tau(t_{i+1} - t_i)})$（τは時定数）式(1)

このＲＣ回路に対する入力電圧と出力電圧は，それぞれ入力ストリームＶと出力ストリームＷとみなされる。入力ストリームＶは，＜time:t, voltage:v＞というレコードの順序列から成り，各レコード（status で示す）は，ある時刻ｔにおける入力電圧ｖを示す。for each は入力ストリームから，個々のレコードを抽出するプログラム構造である。それに続く三行によ

って，出力電圧wと時刻tからなるレコードが生成される。その後，それらの各レコードは，return all という，ちょうどfor each と逆の機能を持つプログラム構造によって出力ストリームWになる。

式(1)に注目してみると，出力Wi+1の決定に際し，入力Viのほかに，一つ前の時点の出力であるWiが関与していることがわかる。つまり，限られた時間の範囲ではあるものの，出力の決定に際して，過去の状態情報を利用する経過依存性の計算を記述していると解釈することができる。ストリームの概念は，このように，従来のフォンノイマン・モデルにおけるメモリセルの代用機能を果たす可能性があるが，まだ研究段階はプリミティブな状態にある。またフランスのツールーズ大学のデータフロー高級言語LAU 18)でも，データベースの更新等を目的として，順路式(path expression)の導入を図っているが，昨年7月の段階ではまだ実装に至っていない。

8.2.6 おわりに

データフロー計算機は，並列処理を計算モデルの中に基本的に含み，フォンノイマン・モデルに比べ，高水準なものとなっている。また，そのためのプログラミング言語も単一割当規則に従う関数型言語であり，数学的基盤を持ち，形式的に整ったものとなっている。経過依存性については，ストリームの概念を用いることで，原理的には，表現可能である。このような性質が，ソフトウェアの整構造化，複雑化の軽減，ハードウェアの構成を，より見通しの良いものとすることなどの利点をもたらすことが期待される。

8.3 ソフトウエア工学との関連性

8.3.1 はじめに

最近のソフトウェア工学の成果は，将来の計算機システムの満たすべき仕様について示唆するものが多く，また，新しい計算モデルやプログラミング言語

の提案も多い。ここでは，主に計算モデルとプログラミング言語について，データフロー・モデルと関連するものを概説する。

プログラミング言語に関する研究は，フォンノイマン・モデルに深く根ざしたものから開始され，多くの経験や試行錯誤を経て，今日に至っている。プログラム言語の研究の目的として，P. Wegnerは，次の4点を挙げている(3)。

① オブジェクト（対象）の定義あるいは，知識表現のための道具の開発

② 大きなシステムをモジュールによって構築するシステマチックな方法を言語面からサポートすること

③ 構造化プログラミング，仕様，検証，テスト，その他，プログラミング方法論を用いて，信頼度の高いコンポーネントの作成技法の開発

以下，これらに対応づけて，ソフトウェア工学の研究の最近の動きと，データフロー・モデル，メッセージ駆動モデルとの関連性について述べる。

8.3.2 オブジェクトのモデル化

プログラム中で扱う対象（object）のモデルとしては，次の3つが考えられる。

① メモリセルのふるまいを抽象化して，オブジェクトとするフォンノイマン・モデル

② パイプラインやストリームをオブジェクトとするストリーム・モデル

③ 能動的にふるまう行為者（agent）をオブジェクトとし，その間のメッセージの交換を行うとするデータフロー，メッセージ駆動モデル

これらのうち，③は，この節の主題である駆動型プログラミングと深く関連するものである。このモデルは，①のオブジェクトである受動的なメモリセルにかわり，マイクロプロセッサなどの能動的な素子をオブジェクトとしており，このようなモデル化によって，より自然に記述できる問題が多くある。この例としては，並列処理，分散処理などのほか，知識表現のための言語などがあり，データフロー計算機研究の強い動機となっている。（8.4のACTOR 理論な

どを参照）

従来のフォンノイマン・モデルに基く高級言語では，オブジェクトは，メモリセルの抽象化であることから変更可能であり，いくつかのプログラム・コンポーネントにより共有されるのが普通である。この結果，あらかじめ，明示されない副作用が生じる。この副作用は，強力な，計算のための機構としても考えられるほか，メモリを有効に使用することを可能としている。

しかしながら，このような機構は，モジュール化を阻害するとともに，適切な制御が難しく，プログラムの虫の元凶ともなり得る。これは，並列処理などのプログラムにおいて，特に問題となる。

オブジェクトのモデル化に際し，このような副作用を無くすことは，フォンノイマン・モデルに基く言語では難しい。一つの方法として，③のような，全く新しいモデルを採用することが考えられる。データフロー・モデルでは，①におけるアサインメント文（assignment by sharing）はなくなり，右辺をコピーして，左辺にわたすという操作（assignment by copying）に置換され，副作用がなくなる。副作用の無さは，並列プログラミングを容易にするばかりではなく，プログラムのモジュール化，仕様の記述，検証などにも，有益であることが知られているほか，関数の性質を有し，数学的な扱いも容易となる。これらは，データフロー計算機を支持するものと考えられるが，ソフトウェア工学の分野では，言語の高級化によって，上記のような性質を実現すればよいのであり，計算モデルとして，新しいものを導入するのは，"radicalなアプローチ"であるとする意見もある(3)。また，データベースの扱いなど，副作用を用いる必要がある分野もあり，最近盛んになった，データフロー言語や，モデルの研究の進展が待たれている。

8.3.3 モジュール化

モジュール化は，大きなプログラムを小さく作る手法として，古くから用いられてきた。ソフトウェア工学は，この目的のために，抽象データ型，カプセ

ル化など種々の手法を提案してきた。これらは，モジュール間インタフェイスをいかに整理し，より抽象的な問題の記述を可能にするかといった問題を扱うものであり，8.3.2のオブジェクトのモデル化とも密接に関連する。このような研究の結果，Alphard(4)やCLU(5)といった新しい言語が提案されている。これらは，データフロー・モデルと直接の関係は無いものの，このような言語の要求する新しい概念が，新しいモデルと良く整合するか否かは，重要な問題である。現在までの，データフロー・モデルや，プログラミング言語の研究は，前述の副作用の無さとも関連して，このような要求にマッチしているように見受けられる。

8.3.4 信頼性の高いコンポーネントの作成

プログラムが，求められる仕様を満たしているか否かの判定は，困難な問題であるが，最近の研究は，このような問題についても，いくつかの示唆を与えている。

既存のフォンノイマン型言語を対象とするプログラムの正当性の証明は，いくつかの困難な問題を提起している。その一つに，プログラムの正しさを論理的に証明しようとする時，プログラムのテキストが，axiom の集合とみなせないというものがある。このため，プログラムの仕様を形式的に整ったassertion として，別個に記述する必要がある。しかし，この場合でも，仕様と実際のプログラムの挙動との矛盾を無くすことは，なかなか難しい。この難点を避けるために，仕様記述のために準備されたformal system を，一部改良して，そのままプログラミング言語として使おうとする試みが為されている。このような言語の例としては，LUCIDがあげられる(5)。 LUCID では，そのテキストは，単一割当規則に基く，equationの集合となっており，これをaxiom の集合として，直接，証明のプロセスを実行することができる。

この研究の狙いは，プログラムの証明であり，データフロー・モデル等の計算モデルとは，直接関係がないが，LUCIDの持つ，副作用の無さ，関数とし

ての性質，単一割当規則などの性質は，データフロー・モデルの持つ性質と非常に近いものである。データフロー計算機のための文章型言語として提案されるもののほとんどが，ＬＵＣＩＤと類似の形式を有していることを考えると，この計算モデルは，このようなソフトウェア工学の要請ともマッチするものと考えられる。

8.3.5 言語構造やモデルの理論的な研究

プログラミング言語や計算モデルを，数学的基盤に基いて，再構築しようとする提案は，これまでにも行われてきた。ＬＩＳＰは，λ計算をもととするこのような試みの結果作られたものである。このほか，一階述語論理をもととするPROLOGなども，プログラミング言語と考えられる(7)(8)。最近では,Backusにより，提案された関数型プログラミングがある(9)。これは，フォンノイマン型の計算モデルの欠点を挙げ，これを解決するものとして，関数型モデルＦＰを提案している。この提案は，λ計算をもとにした関数型言語の優れた性質を，プログラムのモジュール化，副作用，等価性の証明などと絡めて説明している。

以上のような関数型言語やモデルの性質は，データフロー・モデルの性質と共通のものが多い。しかしながら，現時点では，その関連性は明確ではない。これらのより高レベルのモデルと，データフロー・モデルとの結びつきが，どのようなものとなるかは，注目に値するものと考えられる。

8.3.6 まとめ

以上，最近のソフトウェア工学の研究と，駆動型プログラミングとの関連について述べた。ここで述べた点は，多くは，既存のフォンノイマン・モデルと，それに基く言語の問題点に関するものであり，今後新たに構築される計算モデルや言語，アーキテクチャの持つべき性質や，仕様を示していると考えられる。

これら条件を満たすものは，駆動型（データフロー・モデル，メッセージ駆動

モデル）以外のものでもよいわけであるが，現在までのところ，他に，どのようなものが有望なのかは，明らかでない。

いずれにしても，ソフトウェア工学は，フォンノイマン・モデルとの比較をもととして，駆動型モデルとその言語の満たすべき条件について，示していくものと考えられ，実質的な駆動型計算機の研究の多くの部分を占めるものと思われる。

8.4　知識表現をめざす言語との関連性

自然言語処理，パターン認識，知能ロボットなどの人工知能（AI）研究の分野では，人間の持つような知識をいかにして，計算機内部で表現し，推論や学習のために用いるかという研究が精力的に為されてきた。この結果，いろいろな，知識表現や問題解決を目的とした言語が開発されてきた。また，言語の意味（セマンテクス）を与えるための計算モデルも研究されてきた。このような言語やモデルの中で，駆動型プログラミングとの関連で注目されているものとして，C. Hewittにより提案されたACTOR理論と，これに基くプログラミング言語PLASMAがある(10)。ACTOR理論においては，actorと呼ばれる対象（object）と，その間のメッセージのやりとり（message passing）によって，計算が記述される。この計算モデルは，メッセージ駆動型であり，J.B. Dennisらの提案するデータフロー・モデルと極めて近いものといえる。

ACTOR理論においては，演算処理を行うもの（procedureやfunctionなど），メモリ，データを，それぞれ，メッセージを受けとることにより，能動的に機能する手続的な対象，すなわちactorとして統一している。メッセージは，actorに対するオペレーションの要求（必要なデータを含む）や返答（結果を含む）であり，このメッセージがactorへ到着する（これをイベントと呼ぶ）ことによって，オペレーションが実行される。このイベントの順序づけによって，計算が，定義される。actorの集団の中で，メッセージはいくつ

もが同時に伝送されてよく，これによって，並列計算が記述できる。また，actorは，それぞれ，受理すべきメッセージのパターンをいくつか持っており，そのパターンに対応する動作を行う。actor間の関係としては，相互間で直接メッセージの送れるもの（一方が他方を知っている。これをacquaintanceと呼ぶ）と，送れないものがある。actorは，計算の過程で生成されることもある。このような性質のほかに，actorには，その結果が入力のみに依存し，経過依存性の無いpure actorと，経過依存性を持つimpure actorがあり，副作用を必要とするオペレーションも記述できる。

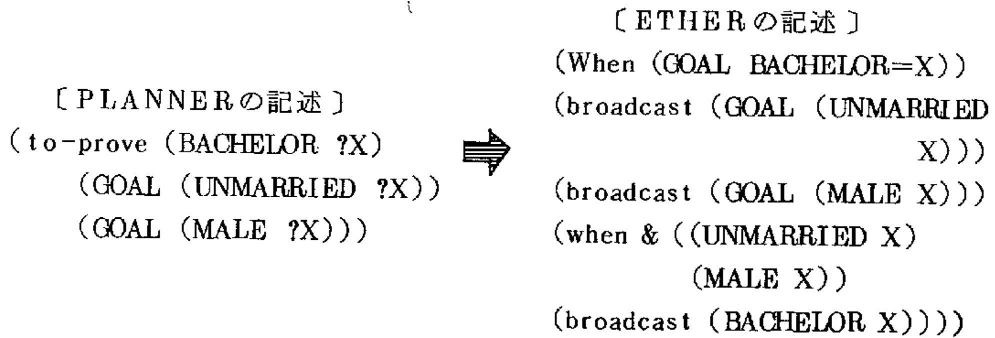
以上のようにACTOR理論は，単に，知識表現を目ざすだけでなく，並列処理や分散処理など様々な計算方式の形式化をも目ざしている。

このほかの知識表現を目ざすものとして，D.G. Bobrow, T. Winogradによる KRLがあげられる(11)。KRLは，フレーム理論に基く表現方法をとり，抽象的な概念や具体的事物を記述するunitをobject（対象）とし，objectの性質や，他のobjectとの関係をunit中のslotと呼ばれる格納場所に記述する。この関係の記述は，unitをノードと考えた場合のアークまたはリンクにあたり，関連するobjectをたどるときに使用される。objectには，手続も付加でき（procedural attachment），TriggerやTrapと呼ばれる機能により，プロセス間で起動をかけることができる。

KRLは，マルチプロセッサ上の実装を考慮しておりsignal mechanismを用いたtrigger event-drivenによるプロセスの起動などが考えられているが，ACTOR理論に見られるような計算モデルとしての提案はない。しかし，そこに見られるオブジェクト指向型の形式化や，プロセス間の起動方式は，メッセージ駆動に近いものが見られる。

以上，あげたほかに，定理証明のようなプロセスを並列実行させようとするものがある。この一例として，W.A. Kornfeldが提案している言語ETHERがある(12)。これは，C. Hewittの作った問題解決用言語PLANNERを基礎

とし，パターン・マッチングの実行を並列化しようとするもので，もととなる定理をメッセージとして伝搬させ，マッチするものからの応答を得て，次々とこの操作を行う。



このような計算方式は，計算モデルとして考えた場合，まだ未整理の点があるものの，メッセージ駆動型と考えられる。

以上，3つの例を挙げたが，これらの言語や計算モデルのもつ，基本的特徴は，次の3点と考えられ，データフローの枠内で，とらえるべき問題と思われる。

① オブジェクト指向型の形式化（モジュール化）

② オブジェクト間のメッセージ交換による交信

③ パターン・マッチングによる起動

しかしながら，知識表現をめざすプログラミング言語においては，その対象は，通常の計算モデルが対象とするものに比べ，より抽象的で，複雑である。また，その主目的とするところも，計算モデルの確立ではない。このため，直ちに，計算モデルとしてのデータフローとの比較は，必ずしも適切とは言えないが，データ駆動，メッセージ駆動のプログラミング，および，それに基く計算機の研究に密接に関連してくるものと考えられる。

8.5 おわりに

駆動型のモデルとそれに関連するプログラミング言語の研究について，現在

行われている並列処理マシンの研究と，将来において，深くかかわると思われるソフトウェア工学や人工知能研究という方向から，眺めてみた。

駆動型のプログラミング言語研究の一つの動機として，従来のフォンノイマン・モデルに基く言語が，十分整った数学的，論理学的な基盤を持たず，巨大さや複雑さに対処する有効な形式的（formal）な手法を活用しにくいことへの反省がある。これは，大きなソフトウェアを開発する場合の見通しを悪くするとともに，ソフトウェアの負担をより軽くする方向へ進歩しようとするマシンのアーキテクチャやハードウェアの進むべき道を示す上でも，極めて不都合である。特に，安価なマイクロプロセッサを大量に作り得るＶＬＳＩ技術の潜在的能力を活用するための方向や，大規模な分散処理システムの構成方法など，現実的な並列処理系のための理論的な基盤などが，既に切実な要求となっており，これらに対する方向づけを与えることも，プログラミング言語やその計算モデルなどの研究にとって，急務となっている。

駆動型の計算モデルは，その実現に対して，多量のプロセッサ，メモリ，および，それらを接続する交信用のバスなどを必要とするが，基本的には，同一部品要素のくり返し構造というＶＬＳＩ化に適した性質をもっている。また，並列処理については，モデル自身の中に基本的に含まれており，逐次処理プロセッサの集合体としてしか並列プロセッサを構成し得ないフォンノイマン・モデルに比べ，より自然な，問題の性質を生かしたプログラミングが可能と考えられる。

現状では，このような原理的に秀れた特徴を十分生かすだけの実用的な形式化が不十分であり，具体的なプログラミング言語や，そのためのマシンを構成するまでには至っていないものの，これまでのフォンノイマン型マシンで得た多くの経験を基に，着実な研究の進展があるものと考えられる。

<参 考 文 献>

〔1〕 Dennis, J.B., "First Version of a Data Flow Procedure Language," Lecture Notes in Computer Science, 19, Springer Verlag, 1974.

〔2〕 Dennis, J.B., and D.P. Misunas, "A Preliminary Architecture for a Basic Data-flow Processor," The Second Annual Symposium on Computer Architecture: Conference Proceedings, January 1975, p126-132.

〔3〕 Wegner, P., Ed., "The Impact of Research on Software Technology", MIT Press, 1979.

〔4〕 Wulf,W.A., et al.,"An Introduction to the Construction and Verification of Alphard Programs", IEEE Trans. on SE, vol. 2, №4, p253-264, 1976.

〔5〕 Liskov, B.H., et al.,"Abstraction Mechanisms in CLU", CACM August 1977.

〔6〕 Aschcroft, E.A. and W.W. Wadge, "LUCID, a Nonprocedural Language with Iteration", CACM, vol. 20, №7, 1977.

〔7〕 中島秀之, "Prologという名のプログラム言語", bit, vol. 10, №12, p67-71, 1978.

〔8〕 Warren, D., "Implementing PROLOG - compiling predicate logic programs", D.A.I. Research Report, University of Edinburgh, №39, May 1977.

〔9〕 Backus, J., "Can Programming Be Liberated from The Von Neuman Style? A Functional Style and Its Algebra of Programs," CACM, vol. 21, №8, p613-614, 1978.

〔10〕 米澤明憲, "ACTOR理論について", 情報処理, vol. 20, №7, p580-589, 1979.

〔11〕 Bobrow, D.G., and T. Winograd, "An Overview of KRL, A Knowledge Representation Language," CSL-76-4, Palo Alto Research Center, Xerox, July 1976.

〔12〕 Kornfield,W.A., "ETHER–A Parallel Problem Solving System" 6th IJCAI, p490–492, 1979.

〔13〕 Winograd, T., "Beyond Programming Languages", CACM, vol. 22, №7, July 1979, p391–401.

〔14〕 Tesler, L.G. and H.J. Enea, "A Language Design for Concurrent Processes, "Proc. of SJCC, vol. 32, p403–408, 1968.

〔15〕 Ackerman,W.B. and J.B. Dennis, "VAL–A Value–Oriented Algorithmic Language: Preliminary Reference Manual," CSG Memo, Laboratory for Computer Science, MIT, 1978.

〔16〕 Ackerman,W.B., "Data Flow Languages," Proc. of NCC, p1087–1095, 1979.

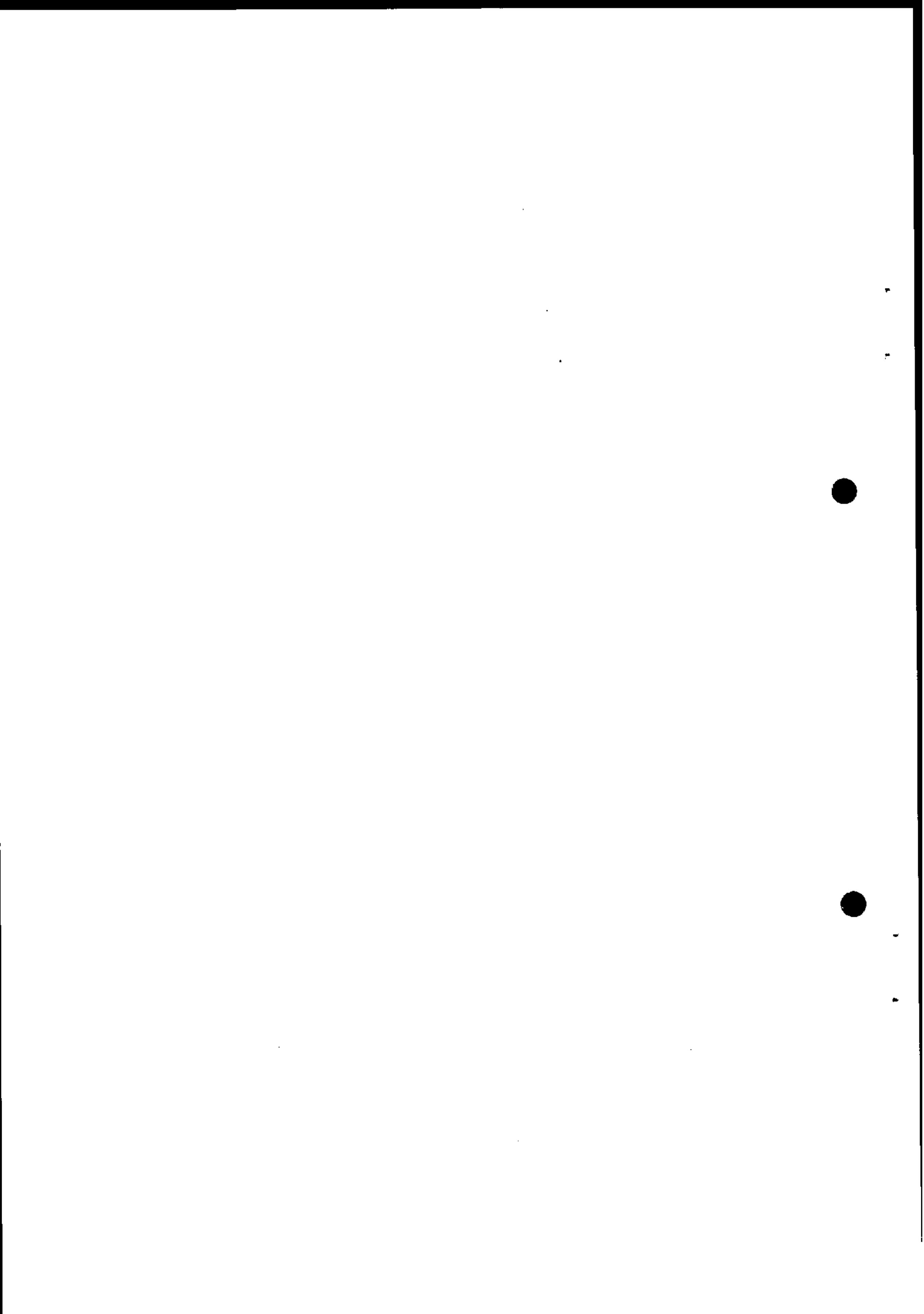
〔17〕 Weng, K.–S., "Stream–oriented Computation in Recursive Data Flow Schemas," TM–86, Laboratory for Computer Science, MIT, 1975.

〔18〕 Comte, D., et al., "Parallelism, Control and Synchronization Expressions in a Single Assignment Language," SIGPLAN Notices, 13, 1, p25–33, 1978.

〔19〕 Arvind, et al., "An Asynchronous Programming Language and Computing Machine," TR114–A, Dept. of Computer Science, University of California, Irvine, 1978.

〔20〕 Bryant, R.E. and J.B. Dennis, "Concurrent Programming," CSG Memo 148–2, Laboratory for Computer Science, MIT, 1978.

第9章　並行プログラミング
(Concurrent Programming)



9. 並行プログラミング

9.1 まえがき

ＯＳは人類がこれまでに作りあげた機構のうちで最も複雑で知的なものであると言われている。ソフトウエアの分野では過去３０年の間その努力の大半をＯＳの開発に向けてきた。その結果，ＯＳの構築方法が近年急速に確立されてきている。単純な機能を行う数多くのプロセスが情報交換をしながら処理をすすめ，全体として複雑な機能を実現するという考え方が基礎になっている。この考え方にもとづくプログラムの作り方を並行プログラミングConcurrent Programming，その目的に沿って作られた言語を並行プログラム言語（Concurrent Programming Language という。*-1

逐次処理を行うノイマン型計算機を効率良く使うために考え出され，ＯＳの基本的なプログラミング技術として現在の計算機を支える技術の１つである。〔Atwood 76〕しかし，単純な機能を行う仮想計算機としてのプロセスを数多く使用し，全体として複雑な機能を実現するという考え方はノイマン型計算機を超越している。現在の計算機よりむしろ，モジュール化，分散処理化，高度並列処理化の必要な将来の計算機に適した技術のように思える。しかし，従来の並行プログラミングは主記憶の共有が前提となっており，共有記憶を持たないシステムにはそのままでは適用できない。並行プログラミングを再構成しなおす必要がある。

並行プログラミングの再構成にあたっては，単なるプログラム技法の再構成

＊　複数のものが物理的に同時に運転されるとき並列parallel，論理的に同時に運転されるとき並行 concurrent という。並行の特殊な場合が並列である。

にとどまらず，より広くシステムの管理と構築の方法の確立を目指す必要があろう。モジュール化と分散処理化の進んだ計算機システムにおいては，計算能力よりシステム全体への融和能力の方が重要である。計算方法のプログラミングよりシステムの管理と構築方法のプログラミングの方が重要なのである。

こうした方法論は，プログラム言語に正確に組み込んではじめてプログラム言語で安心して確実に使えるものになる。〔Brinch Hansen 79〕。そこで，大規模な並行処理システムの管理と構築の方法を，簡潔，明確，正確に言語として実現することが並行プログラミングの中心テーマとなる。

本章は，大規模並行処理システムの管理と構築の方法のプログラム言語化を目的として，従来の並行プログラミングを整理する。まず9.2で並行プログラミングが過去取扱ってきた問題と将来取扱うべき問題点を明らかにする。続いて9.3で並行プログラミングの適用分野の特長と問題点について述べる。9.4では最も基本的な同期と通信の機構を言語化に注目して整理し，9.5で言語の設計要件についてコメントを加える。

9.2 歴史的概観

計算機技術は，ハードウエアの高価な時代とソフトウエアの高価な時代を経て，システムの高価な時代〔Winograd 79〕に入っている。この時代区分に従って並行プログラミングを概観する。

9.2.1 高価なハードウエア

初期のハードウエアは非常に高価だったので，その使用効率をあげ経済性を高めるのが重要な課題であった。

入出力の速度は計算速度に比べて非常に遅い。そこで入出力動作と計算動作を並列に行なわせるために非同期データチャネルと割込のハードウエアが開発された（なお，並行プログラミングのために開発されたハードウエアは現在までのところこの2つだけである）。cpuの時間が余ることになった。この余っ

た時間を有効に使うために，複数のプログラムを同時にメモリに入れておき，1つが入出力待ちになると他のプログラムを走らせることが考え出された。1つのプログラムを時間に依存しない動作をする非同期プロセスに分けた。これを使って必要とする資源が競合しないプロセスを多重処理するシステムや，入出力の多いことを積極的に利用する会話処理型システムが作られ成功した。システムの性能は価格の2乗に比例する（Groschの法則）と言われ，システムの巨大化と多重化が進められた。1960年代半ばになると，いたずらに，MULTICS，OS360などの現在使用中の巨大システムが作られ始めた。目先の効率をあげるために，アドホックな方法でしかも低レベル言語でシステムを作るという禍根を残してしまった。様々な言語で書かれたプログラムの多重処理を行なう際の信頼性をあげるために，様々なレベルでのチェックが必要である。その結果プロセスの実現コストを非常に高いものにしてしまった。

9.2.2 高価なソフトウエア〔Brinch Hansen 79〕

〝OS360ではリースのたびに常に1000個のバグが存在する〟と言われ，未整理な概念でOSを作ることの危険性が痛感された。アドホックな方法を整理し安全な方法論を確立することが課題であった。

まずプロセス同志のインタラクションを整理し，理解しやすいようにすることに力が注がれた。Dijkstraは並行プロセス同志の一般的な同期の概念を臨界領域critical region，センフォsemaphore，P－V操作の形に整理し〔Dijkstra 68a〕，それを用いてモデルOS，THE〔Dijkstra 68b〕，〔Mckeag 76〕を作りあげてみせた。THEは実際にはアセンブラでインプリメントされていたが，並行文concurrent statementを導入し，文が並行に実行されるという概念をプログラム言語で記述してみせた〔Dijkstra 68a〕。

概念が整理されると，その概念をより厳密にし，記法化してプログラム言語化することに力が注がれた。プロセス同志が協調動作できるために，共有変数

を臨界領域に関係づけ，その内では同時には１つのプロセスだけがアクティブになれるようにした。複雑な同期を実現するために条件付臨界領域conditional cvitical vegionが提案された。

一方Dijkstraはプロセスのインタラクションをはっきりさせるために，共有データを中心に関連ある操作を１つのモジュールまとめることを示した。〔Dijkstra 71〕

以上の条件付臨界領域と共有データ中心のモジュール化の考え方（抽象データ・タイプ）が統一化され，モニタ（monitor）〔Hoare 74〕，〔Brinch Hansen 73　やセクリタリ(secretary)〔Dijkstra 71〕という概念になった。Brinch Hansenはモニタの概念をPascalに導入してConcurrent Pascal〔Brinch Hansen 75〕を作り簡単なＯＳを書いてみせた。

こうして，ユニプロセッサ上のプロセス同志のインタラクションの問題は，抽象データタイプとモニタによって解決されたと言われている〔Denning 77〕。

9.2.3 高価なシステム

永年の課題であったユニプロセッサ上の並行プログラミングの問題は一応解決されたが，極度なハードウエア・コストの低下と分散システムの普及の結果，並行プログラミングの分野に新たな問題が生じてきた。

○低廉化したマイクロ・エレクトロニクスを有効に生かすようなプログラミングはどのようなものであろうか。

○分散システムに適したプログラミングはどのようなものであろうか。

最初の問題は，並列処理の問題であり，計算の各ステップを単位とした並列化（並列計算）〔Uchida 80〕と，モジュール単位の並列化に分けられる。本章の対象はモジュール単位の並列化と分散システムである。

これに関してWinogradは，″計算機は複雑なシステムの１要素にすぎない。近い将来，考えられるありとあらゆる装置にマイクロ・コンピュータが内蔵され，それらが何らかの形でネットワークに組み込まれる。そこではソフトウエ

アとハードウエアのレベルで，パッケージを集めたり，修正したり，既存のパッケージの動作を説明することがプログラミングの主流になる″〔Winograd 79〕ことを指摘している。このようなシステムは当然のことながら，リアルタイム・システム，オペレイティング・システム，分散処理システム，並列処理システムなどの様々な側面を持つことになる。社会システムとでも呼べるような大規模で広域化したシステムのプログラミングや構築論が将来の並行プログラミングの扱う対象となるであろう。

並行プログラミングが今後扱う問題に対する現時点でのアプローチは，初期の頃と同じようにアドホックである。概念の整理，記法化，言語化の過程を経て現在の並行プログラミングができあがったように，今後もこの過程を経る必要があろう。

並行プログラミングが今後扱うべき問題点を以下に列挙する。

- プロセスの通信と同期の再点検
- モジュールの分割と割当て
- 信頼性
- エラー・リカバリ，再構成
- 保　護
- 抽象化
- 性能の予測
- システムの構成法
- データフロとの関連

9.3 適用分野

並行プログラミングが今後対象とするシステムは様々な側面を持つことを9.2.3で述べた。ここでは，その側面の特長を述べる。現在の並行プログラミングのアプリケーションに関するコメントにもなっている。〔Grand 79〕

9.3.1 リアルタイム・システム

リアルタイム・システムは並行プログラミングの最も簡単な応用分野である。周期的や割込で走るプロセスが中心で，その処理は通常非常に簡単である。それゆえ，強力な言語のメカニズムは不要である。高級な機構は通常効率が悪く，リアルタイム・システムの目標である割込不応時間Latencyが100 μsec〔Lycklama 78〕を越えてしまう。

リアルタイム・システムは非標準的なデバイスを扱うことが多く，そのハンドラが言語上で記述できねばならない。〔Wirth 77d〕

9.3.2 オペレイティング・システム

通常ＯＳは，正しいことが仮定されたカーネルKernel〔Popek 78〕と，誤り易く時には悪意さえ持っているアプリケーション・プログラムから成る。カーネルは，その上でのアプリケーションの展開方法を規定し，システムの構築方法を提供するプログラムである。その方法に従って構築されたプロセスを保護する。

Concurrent Pascalなどの並行プログラム言語は，ランゲージ・カーネルLanguage Kernelとコンパイラによってプロセス同志の保護が完全にできる。アプリケーションの展開方法を言語の文法で明確に定義できるので，プログラム言語からのアプローチが良いように思える。しかし，完全な言語の設計は不可能であり，保護が破れたり，アプリケーションのポリシーを強く制限することになりかねない。例えばAda〔Honeywell 79a~b〕〔Horeywell 79b〕では，どのプロセス（Adaではタスク）からでも特権操作が行なえるし，ポインタを使ってどこにでもアクセスできてしまう。また，Concurrent Pascalで作られたSolo〔Brinch Hansen 76a~c〕上のアプリケーションは特別なSequential Pascalに限られている。

ケイパビリティ（capability）〔Hsiao 79〕などの保護機構をプログラ

ム言語に導入しなければならない。

9.3.3 分散システム

分散システム用のプログラム言語は，リアルタイム・システム的な性格とOS的な性格をあわせ持たねばならない。

分散システムの上位のプロトコルでは，メッセージがデータ・タイプを持っているのに対し下位のプロトコルではタイプがなかったりあっても弱い。中位のプロトコルではタイプをとり除いたり，タイプの下のデータにアクセスすることが必要である。そうでなければ，CRCチェック，パケットへの分解，や組立てができない。下位のレベルはタイプの弱いリアルタイム・システム的な言語，上位のレベルはタイプの強い言語が望ましい。

ネットワーク上のアプリケーション・プログラム，例えば複数のノードにまたがるプログラム，ではモジュールの壁を越えてのアクセスを厳格にチェックできなければならない。他ノードの誤りがシステム全体のダメージにつながるおそれがある。通信の信頼性の悪さや走行中のシステムの消滅を考慮しなければならない。従来の並行プログラム言語では，プロセス間のインタフェイスは完全であるとの仮定があったが，分散システムではこの仮定は通用しない。誤り検出，回復，保護等のOS的性格が言語にも必要である。

9.4 プログラミングの概念

9.3で並行プログラミングが対象とするいろいろなシステムの性格を述べたが，ここではそれらに共通で最も基本的なプロセスの並行性を実現するための機構について述べる。

複数のプロセスに互いに意味のある並行動作をさせるためには，その間の相互作用が必要である。次の2種類の相互作用を考えなければならない〔Bryant 78〕。

○通信　Communication　：　データの伝達

○同期　Synchronization：　並行動作の部分的実行順序の保障（順序化precedence constraint）とプロセス同志で安全に資源を共有させるために他の並行動作が実行されないことの保障（相互排除mutual exclusion）

通信と同期の基本操作はこれまでに数多く提案されている。ここでは高級言語化に注目して次の4種類に分類する。

- 同期（手続）指向型
- 通信（メッセージ）指向型
- 仕様指向型
- コルーチン指向型

これらの相互関係およびその中の具体的な機構の関係を図9－1に示す。図中にアルファベットが書かれたものはその機構を導入した代表的な並行プログラム言語である。

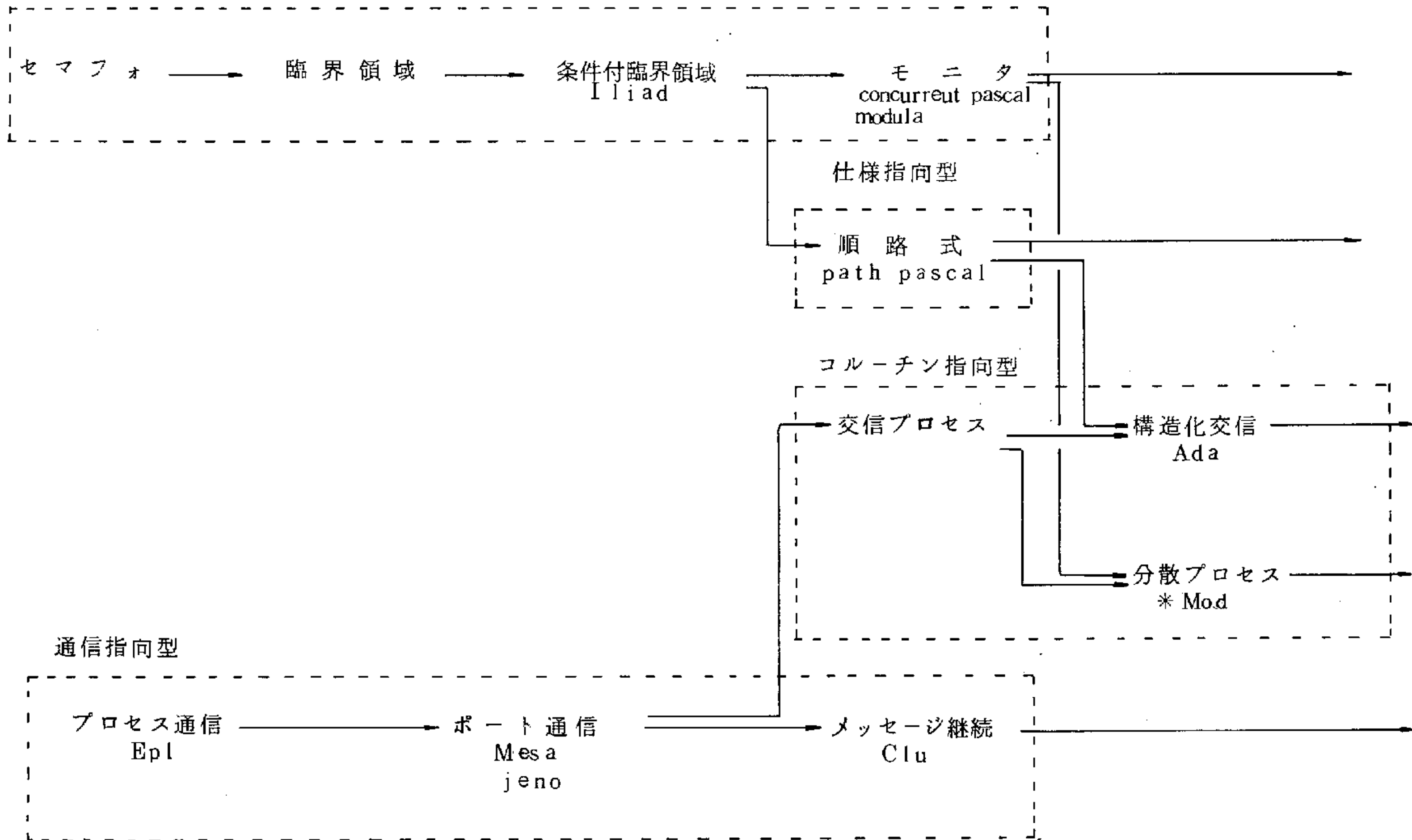


図9－1　並行プログラミングの系譜

9.4.1　同期（手続）指向型

フラグに対する〝参照－判断－変更〞が一時に一つのプロセスに限られれば，そのフラグを用いて複数のプロセスの同期をとることができる。効率良く実現できるので，昔からよく研究され使われてきた。つい最近までは，並行プログラミングと言えばこの型そのものであった。

(1)　セマフォ（ semaphore ）〔Dijkstra 68a〕

古くからある簡単な同期の機構である。非負の値をもちプロセスの待ち合わせのキュが附随した制御用の特別な変数のことをセマフォという。セマフォには２つの操作Ｐ／Ｖ（ wait／signal ）が定義される。

```
S : Semaphore(initial value);
Procedare V (S:semaphore);
begin [ S := S+1 ;
        if s≧o then wake up
a waijing process on the queue] end;
Proceduve  P(s;semaphore);
begin [ s := s-1 ;
        if s<o then enqueue and block
the process the quoue] end;
ここで〔……〕は不可分操作
```

図９－２　Ｐ，Ｖ操作の定義〔Andler 78〕

セマフォの初期値の与え方で，いろいろな目的に使うことができる。例えば，初期値１のとき相互排除用，０のとき順序化用。

しかし一般的すぎて，セマフォの使用方法や意味を理解するのが困難である。Ｐ，Ｖ操作が構造化されていないので臨界領域critical regionになっているか否か，セマフォが保護するデータはどれなのかがわからない。さらに悪いことには，Ｐ，Ｖ操作がプログラム全域に散乱するのを防ぐことができない。機構としては十分プリミティブで強力であるが，プログラミングや検証のためには不十分であった。

```
var  nr-of-pueuing-portions:semaphore(o);
     nr-of-empty-portions:semaphore(n);
     buffer-manipulation:semaphore(1);
procedure send;
begin  produce next portion;
       P(nr-of-empty-portions);
          P(buffer-manipulation);
             add portion to buffer;
          V(buffer-manipulation);
       V(nr-of-queuing-portions) end;
procedure receive;
begin  P(nr-of-queuing-portions);
          P(buffer-manipulation);
             take portion from buffer;
          V(buffer-manipulation);
       V(nr-of-empty-portions)  ;
       process portion taken  end
```

図9－3　セマフォ（リング・バッファ）〔Andler 78〕

図9－3は，リング・バッファをセマフォを使って記述したものである。ここでは3つのセマフォが使われている。Buffer－manipulation はバッファへのアクセスの相互排除用にNr－of－queuing－portions とNr－of－empty－portions は send と veceive の同期用に用いられている。

(2)　条件付領界領域　conditional critical region〔Brinch Hansen 72〕,〔Brinch Hansen 73〕

相互排除用に使われるセマフォは常に次のような形で使われる。

```
var  M:semaphore(1)  ;
begin ……;P(M);  S;V(M);…… end
```

P (M) ；S；V (M) を一つの文にする。

さらに，共有変数を臨界領域Mに関連づける。コンパイラでは関連づけられた臨界領域内でのみその変数にアクセスしていることをチェックすれば良い。

```
var M: shared record  a;…… end;
begin …… ;regin M do begin ……;f (a) ;……end;……end
```

これだけでは，プロセスの処理の順序化が実現できない。そこでregion文を拡張し，臨界領域へ入るための条件が記述できるようにする。

```
var M: shared T;

begin …… ; region M  when exp do s;……end
```

ここで，同一共有変数名（臨界領域名）をもつ臨界領域内で同時に2つ以上のプロセスがアクティブにならないことを保障しなければならない。1つのプロセスがアクティブになると他のプロセスはそのプロセスが同じ臨界領域を出るのを待つ。プロセスが臨界領域から抜けるときには，他のプロセスが臨界領域に入れるかどうかを調べるためにその臨界領域に入るのを待っているプロセスのwhen句を再評価する。同期排除領域と共通変数の関係も明確になった。しかし条件式をくり返し評価する必要があり，実行時の効率が良くない。またFIFO以外のスケジューリングを実現することができない。

リング・バッファは条件は臨界領域を用いて図9－4のように記述される。

```
var  b: shared record    h, inx, outx: integer (o) ;
                                  buffer:array [o.. buffsize−1] of
                                              message end;

procedure send  (m:message) ;
  begin  region b when n<buffsige do
                          begin buffer  [inx] :=M;
                        inxx := (inx+1) mod buffsize;
                        n:=n+1  end
```

```
procedure receive (var m:message) ;
  begin region b when n>o  do
                        begin  m:=buffer [outx] ;
                               outx:= (outx+1) mod buffsize;
                               n:=n+1  end
end;
```

図9−4 条件付臨界領域（リング・バッファ）

(3) モニタ monitor [Hoare 74], [Brinch Hansen 77b]

条件付臨界領域は効率よく実現できない。また，同一名の臨界領域がプログラム全域に散乱するのを防ぐことはできない。Simula 67のクラスにならい，条件付臨界領域をそれぞれ手続きにし，臨界領域名が同じものを1ケ所にあつめモニタと呼ぶ。同一モニタ内では同時に2つ以上のプロセスがアクティブになれない。その内ではイベント・キュに対するsignalとwaitで同期をとる。

プロセスがモニタに入るときに必ず通過するメインキュQv，モニタ内でアクティブになるのを待つためのサブキュQv′，とイベントを待つためのイベントキュQeiを用いてモニタの動作を説明する。

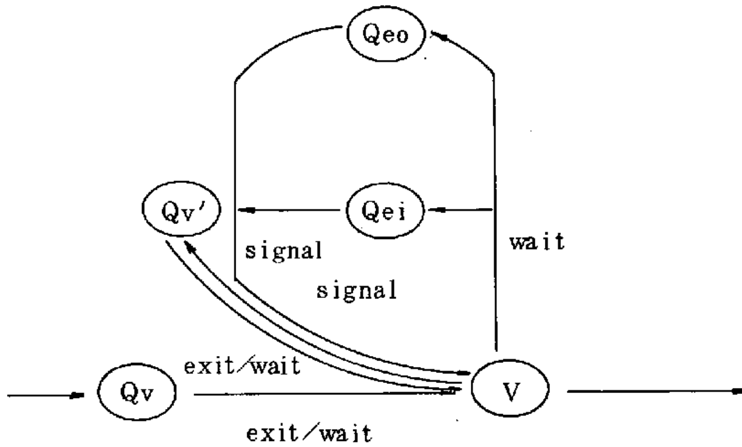


図９－５　モニタのキュー

signalに出合うと，そのプロセスはQv′に入りイベントに対応するQeiからプロセスを取り出してアクティブにする。アクティブになるプロセスがなければ，Qv′の先頭のプロセスを取り出してアクティブにする。waitに出合うと，対応するイベントが発生していれば，そのまま処理を継続し，未発生ならば対応するQeiに入る。waitでプロセスがインアクティブになるとき，あるいはモニタから出るときには，Qv′，Qvの順で先頭のプロセスを探し出してアクティブにする。

イベント待ちのプロセスに対し，signalを発したときの制御の移動は，図９－６のようになる。〔wirfh 77c〕

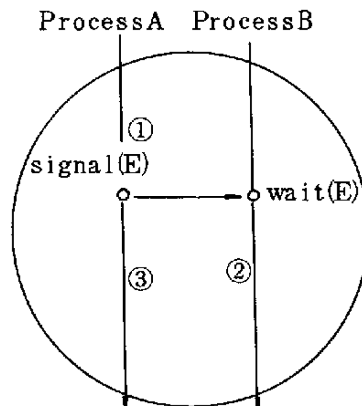


図９－６モニタでの制御の移動

```
wonitor    ringbuffer;
    var      n, inx, outx:integer (o) ;
             nonempty, nonful :eventqueue;
             buffer:array [o. buffsize-1] ofmessage;
    procedure  send (m:message) ;
    begin    if   n=buffsize then wait (nonfull) ;
             buffer [inx] :=m;
             inx:= (inx+1) mod buffsije;
             n :=n+1;
             if  awaited (nonempty)  then signal (nonempty)
      end;
    procedure  receive (var m:message) ;
        begin    if n=o then wait (nonempty) ;
                 m:=buffer [outx] ;
                 outx := (outx+1) mod duffsize
                 n :=n-1;
                 if awaited (nonfull) then signal (nonfull)
          end
    begin…end ;
```

図９－７　モニタ（リング・バッファ）

モニタを用いれば，リング・バッファは図9－7のように記述される。

ところで，モニタの実証者の1人であるBrinch Hansenのモニタでは Qv' が存在しないために，signal はモニタ手続中で唯一回しかもモニタ手続の最後でしか使えない。

同期を基本にした並行プログラミンは，セマフォに始まり，条件付臨界領域を経て，クラスと抽象データ・タイプの概念をとり込みモニタとして完成した。モニタの概念を基本にして，Concurrent Pascal〔Brinch Hansen 75〕,〔Brinch Hansen 77b〕,Csp／k〔Holt 78〕, Modula〔wirth 77a〜c〕,〔wirth 77b〕,〔wirth 77c〕,等数多くプログラム言語が作られている。そのような言語を用いると，簡単なOS，例えば1人用のOS，だと1300行ほどで作れる〔Brinch Hansen 76a〜d〕,〔Brinch Hansen 76b〕,〔Brinch Hansen 76c〕,〔Brinch Hansen 76d〕。

しかし，その後の研究の結果，モニタは，未だ不十分であることがわかってきている。モニタは同期と通信に関する抽象データタイプであり，階層的にシステムを構築する際には〔Dijkstra 68b〕,〔Lister 77b〕，抽象化が多層になり，モニタ中で他のモニタをコールするという状況が表われる(モニタのネスティング)〔Lister 77a〕,〔Haddon 77〕。その際モニタの相互排除を解かなければネスティングの奥の方で，すでにブロックされたモニタをコールし，デッドロックに陥いる。相互排除を解けばモニタ性が保障されなくなる。すなわち，モニタには多層の抽象化機能が欠けており，大規模システムの構築用としては不十分である〔Keedy 79〕。

9.4.2 通信(メッセージ)指向型

通信の考え方は古くからあり，その起源は定かでない。共有変数や共有領域を用いた場合に比べ効率が悪いので，特殊な状況下以外では使用されてこなかった〔Hunt 79〕。ごく最近になって，共有記憶を必要としない点が注目され，分散システムのプログラミングとして脚光を浴びてきた。

(1) プロセス対プロセス通信

メッセージ通信の最も素朴な形は，プロセスがそれぞれ受信バッファを持ち，送信プロセスがメッセージをそれに直接送り込む方法である。指令や返事の意味を持ったメッセージが同じバッファに送られてくる。両者は，通信の形式が少し異なるので，基本操作としては次の4通りを備えることになる〔Lagally 78〕。

① Send-order (order, idn, proc):

orderをprocに送る。返事がどの指令に対するものか区別するためにシステムは，一意な識別番号を生成しidnにその値を入れる。

② receive-order (buffer, idn, proc):

指令が入ってくるまで待ち，指令の内容，指令の識別番号，指令発信元アドレスをbuffer, idn, procに入れる。

③ Send-Answer (answer, idn, proc):

idnで識別される指令に対する返事をprocに返す。

④ Receive-Answer (buffer, idn):

返事が入るまで待つ。返事の内容と識別番号がbuffer, idnに入れられる。

これだけの基本操作で指令の受信順序とは異なった順序で返事を返す（スケジュール）ことができる。指令発信側では複数の指令が出せても返事が順序通りに戻ってこないので，これだけでは，プログラミングが非常に繁雑になる。選択的に受信する機能が必要である。

⑤ Receive-Answer-Strict (buffer, idn):

idnで指定する指令に対する返事を待つ。

セマフォの機能をするプロセスは，図9－8のように記述される。

```
process   semaphore ;
   var    idn:identifier;
          proc:processid;
          order:(P,V);
          counter:integer(n);
          g : queue of (idn,proc);
  begin
       loop
          receive-order(idn,order,proc);
          case order of P:begin  n:=n-1;
                                 if n<0 then enqueue(idn,proc)
                           end;
                        V:begin  n:=n+1;
                                 if n≧0 then
                                    begin dequeue(idn,proc);
                                          send-answer
                                          (idn,"ok",proc)
                                          end
                          end
          end case
       end loop
process   user;
   ...
  begin   ...
          send-order(x,P,"semaphore");
          receive-answer(x,y,s);
          ...
          send-order(x,V,"semaphore");
          receive-answer(x,y,s);
  end
```

図9－8　プロセス間通信（セマフォ）

基本操作としては十分にプリミティブであるが，プログラミングや検証のためには不十分である。指令の受信とスケジューリングのためのモニタの機能を全てのプロセスがかかえ込むことになる。メッセージのデータ・タイプの不一致をあらかじめコンパイラで検出することができない〔Hunt 79〕。各プロセスが出せる指令数を制限しないと資源としてのバッファが尽きてシステム全体がデッドロックに陥いる可能性がある〔Brinch Hansen 73a〕。

(2) ポート port

プロセス対プロセスの通信では1つのバッファが多目的に使われ，プログラミングや検証が困難であった。プロセスのメッセージの入口と出口を目的ごとに分け，各々に名前をつける。ポートと呼ぶ。各ポートには，それが送信，受信することのできるメッセージのタイプを付ける〔Balzer 71〕。

他のプロセスに機能を提供するプロセスでは関連するポートを移出 export し，機能を使用するプロセスではポートを移入 import する。export と import によってコンパイラは物理的なバッファを作成し，対応するポートのタイプがチェックできる。export，import，される port 以外は他のプロセスから隠すことができるので，モジュラリティが向上する。

export p (a_1 : t_1 ,………)

import q (b_1 : t_1 ,………) from process

通信の基本操作は次のようになる。

① Send - order (port，proc) :

port の指令を proc に送信する。

② Receive - answer (port) :

port に返事を受信するのを待つ。

③ Receive - order (port，proc) :

port に指令を受信するのを待つ。送信元のアドレスが proc に入る。

④ send - answer (port, proc):

port の返事を proc に送信する。

このままでは，複数のポートに対し，そのどれでも良いから指令を受信したらそれに応じた処理をすることができないので，Receive-orderを次のように拡張する〔Lauer 78〕。

③′ Receive - order (set-of-port, port, proc):

set-of-port で指定したポートの１つに指令が送られてくるのを待つ。指令を受信するとその１つをとり出し，受信したポートのアドレス（名前）と相手プロセスのアドレス（名前）をそれぞれ port-proc に入れる。

これだけの基本操作を用いて簡単なリソースマネージャは図９－９のように記述される。リソースを管理するために必要なローカルデータと複数のポートからの指令を待ち，その処理をするためにループが必要である。

```
 var      m  : message;
          po : portid;
          pr : processid;
          S  : set of portid;
begin loop  wait-order(S, po, pr);
            case po of
                  port1:{algorithm for port1};
                  port2:begin if resource exhausted then
                                    S:=S-port2;
                              send-answer(po, pr);
                              {algorithm for port2} end;
                           ...
                  portk :begin ...
                                    S;=S+port2;
                              {algorithm for portk } end
            end case ;
      end loop ;
end
```

図９－９　　ポート通信〔Lauer 78〕

(3) メッセージ継続 message continuation〔Liskof 79〕

今までは，一連の交信の間の関係を保持するために，メッセージを指令と返事に分け，交信に識別番号や発信元のプロセス名を伴ってきた。これらはメッセージ継続という考え方で統一的に扱うことができる。すなわち，送信にあたって，返事を期待するときには返事を受信するポートリストのアドレスを付けて送信する。

① send(args)to ports 〔reply to portr〕：

ports に（args）と返事を受信するためのポートリストのアドレス portrを送る。送信が終了すれば次の処理に移る。

② receive on portlist when C_i(formal args)〔reply to formal port〕: S_i;

⋮

when Failure(S:string):sfailure end

メッセージがポートリスト portlistにすでに到達していればそのうちの1つを，Ci(formal-args)に，またそのときの通信用ポートリストのアドレスをformal-port に受けとり，対応する処理 Si を行う。相手ポートが存在しなかったり，通信エラーが発生すると通信ポートFailureにFailure('……')が返される。

メッセージ指向型では，アドレス空間の制約から，プロセスがモジュール化の単位になっている。プロセス間の交信操作としては十分にプリミティブである。

9.4.3 仕様指向型

モニタは同期に関する操作をモニタに閉じ込めることに成功したが，実際の同期操作はモニタ内に散乱しており検証は容易ではない〔Lagally 78〕。同期に関する仕様から同期命令が機械的に生成できれば検証が楽になる。そこで共有データにアクセスするための操作の集合に対し，操作の許容可能なシーケンス（順路）を定義する。順路式path expression という。この式は，順

序化″;″，選択″,″ くり返し″*″，並行″{ }″のオペレータを用いて正則式として表現できる〔campbell 74〕。共有データへの操作が順路式で定義されたものでないときには，実行可能な状況になるまで待たされる。

順路式を用いれば，非手続的に同期が表現でき，使用者は同期の記述から解放される。さらに，順路式をタイプとしてデータに附随させることができ，モジュラリティが良くなる。しかし記述能力が低く，スケジューリングを含んだ同期問題の記述が困難である。

記述能力を高めるためにいろいろな努力がなされてきたが，ここでは，最もエレガントで強力だと思われる述語順路式 predicate path expresstion〔Andler 79〕の要点について述べるにとどめる。順路式の欠点は，スケジューリング，すなわち，操作が実行できる条件が簡明に記述できないことであった。臨界領域にも同じ欠点があったが，条件待ちのために when 句を追加し，条件付臨界領域にすることで解決できた。これと全く同じことを順路式に適用し，ある操作が実行可能であるための条件（述語）を op〔predicate〕の形で書く。predicate を履歴用変数の表われる式に限ると，順路式がプログラムに逆戻りする危険性が避けられる。しかし，一連の処理を必要以上に寸断して操作にするという問題点が残されている〔Yonezawa 79〕。

```
type     ringbuffer〔buffsize：integer〕＝
   var   n, inx, out ： integer(0);
         buffer ： array〔0‥buffsize－1〕 of message;
   path  (send〔n＜buffsize〕 | receive〔n＞0〕)* end;
   export procedure  send(m：message);
               begin buffer〔inx〕：=m;
                     inx：=(inx+1)mod buffsize;
                     n：=n+1
                end;
   export procedure  receive(var m：message);
               begin m：=buffer〔qutx〕;
                     outx：=(outx+1)mod buffsize;
                     n：=n－1
                end
endtype
    ---
var      buf：ringbuffer(10);
         x  ：message;
begin    ---
         buf.send(x);
         ---
         buf.receive(x);
         ---
end
```

図9－10　述語順路式（リングバッファ）

9.4.4 コルーチン指向型

今までの型では，制御とデータを互いに独立なものとして扱ってきた。ここでは，両者を同一化し，簡明な方法を求めようとする最近の試みについて述べる。

プロセス間で制御がバトンタッチされるという共通点を持っており，コルーチン指向型と呼ぶことにする。

(1) 交信プロセス communicating sequential process〔Hoare 78〕

通信指向型は本質的には，ノーウエイト型で，4.2で述べたようなOSの機能を各プロセスが抱え込む必要がある。本来の問題の構造に比べかなり複雑なプログラムになる。交信方法と制御に工夫をこらしてプログラムの構造を問題の構造に近づける最初の試みは，Hoareによってなされ，CSP(Communicating Sequencial Process)と呼ばれている。

① 各コマンドの実行順は，ガーデッドコマンド〔Dijkstra 75〕で制御する。〔……〕は選択的実行，〔……〕はくり返し実行，〔Com 1 || Com 2 ||…〕は並行実行を意味する。

② プロセス交信用のコマンドは次の2つである。

souce ? variables	:	プロセス souce から variables に入力する。
destination!expressions	:	プロセス destination に expressions の値を出力する。

交信するプロセスは，データの構造まで含めて，入力と出力のコマンドが一致するまで待たされ，一致した時，実際の交信が行なわれる。

③ ガード部分に入力コマンドや出力コマンドが表われてよい。

アイデアの提案であり，そのインプリメントは定かでない。非決定的な交信の必要性と，そのための記法の提案は注目に値する。

バッファの機能をするプロセスは図9－11のように記述できる。

```
ringbuffer::
        n, inx, outx : integer ; n := 0 ; inx := 0 ; outx := 0 ;
        buffer : (o‥buffsize - 1) message ;
        ＊〔n < buffsize ; sender?buffer(inx) → inx := (inx+1) mod
                                                        buffsize;
                                                      n := n + 1 □
          n > 0 ; receiver!buffer(outx) → outx := (outx+1) mod
                                                        buffsize;
                                                      n := n-1 〕
sender::
        x : message ;
        ---
        ringbuffer!x ;
        ---
receiver::
        x : message ;
        ---
        ringbuffer?x ;
        ---
```

図9－11　交信プロセス（リングバッファ）

send, receive の要求はいつくるか予測できない。非決定的処理ができなければ，入力を待ちつづけたり，出力の終了を待ちつづけることになる。そこでガーディドコマンドが威力を発揮し，実行できる条件が整ったコマンドに対し，入出力処理が行なわれる。

(2)　分散プロセス distributed process〔Brinch Hansen 78a〕

交信プロセスは言語と計算のモデルである。交信プロセス同志は相手の名前を知っておかねばならないし，他のプロセスをスケジュールするプロセスが記述できない。現実のシステムの記述のためには致命的な欠陥を持ってい

る〔Horeywell 79b〕。これらの問題点を解決するためにBrinch Hansenはモニタを次のように拡張し分散プロセスdistributed processと呼んだ。

各モニタ手続に対し十分な数のプロセスを割り当てる。このプロセスは，各モニタ手続の入口にある待ち行列で，他のプロセスからの要求を待ち受ける。他のプロセスはこの手続をコールすると，パラメータが送信され，結果を受信するまで待たされる。入口の待ち行列の先頭のプロセスはパラメータを取り込み指定のモニタ手続の処理をする。処理が終了すると結果を要求元に戻す。再びモニタ手続の待ち行列に入って次の要求を待ち受ける。

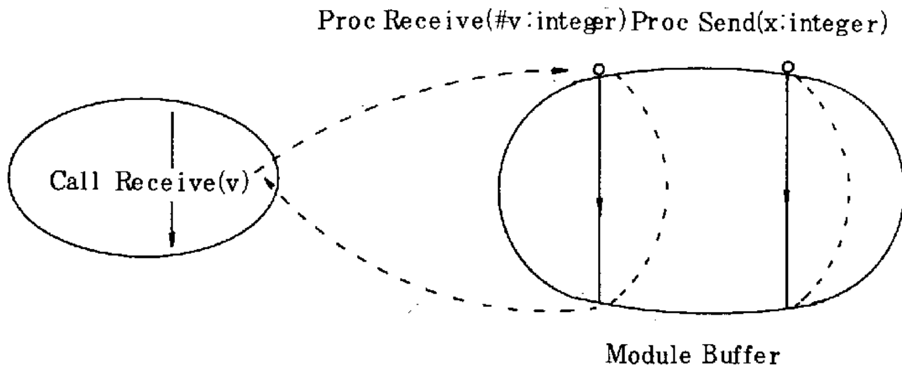


図9－12　分散プロセス

リングバッファの機能をするモジュール(〔Brinch Hansen 78a〕ではprocessという)はモニタと同様に次のように記述される。

同期指向型の成果をふまえた現実的な解であるように思える。

```
process   ringbuffer ;
    n, inx, outx : int
    buffer : array [0..buffsize-1] message
    proc  send (m:message);
        when  n<buffsize : buffer[inx] :=m
                           inx:=(inx+1)mod buffsize
                           n :=n+1
    proc  receive(#m:message);
        when  n>0  :  m:=buffer[outx] ;
                      outx:=(outx+1)mod buffsize
                      n:=n-1
process user;
    ---
    call ringbuffer.send(x)
    ---
    call ringbuffer.receive(x)
```

図9－13　分散プロセス（リングバッファ）

(3)　構造化交信 Structured Communication 〔Honeywell79a~b〕〔Horeywell 79b〕
通常の交信では，このような形式がよく表われる。

```
process  Pm;
   port   p( … ), q( … );
   begin   ---
       send-order(p, Ps);

       receive-answer(q);
       ---      end
```

```
process  Ps;
   port    p( … ), q( … );
   begin  ---
       receive-order(p, Pr);
              S
       send-answer(q, Pr);
       ---          end;
```

図9－14　非構造化交信

send-order と receive-answer，receive-order と send answer において，send と receiveの順序関係とパラメータの対応を見つけるのは困難である。交信を構造化し，前者を call P，後者を accept P do S end におきかえる。図9－14の交信は図9－15のようになる。

call P(…)を行なったPmは，相手Psが結果を返して来るまで待たされる。一方，accept に達したPsは，他から要求が来るまで待たされる。

```
process  Pm;                                      process  Ps;
import P(input param# output paran)fromPs  export P(input param# output param);
  begin  ---                                      begin  ---
         call P(…);                               accept P(…) do S end;
         ---  end                                        ---  end
```

図9－15　構造化交信

両者が揃ったときパラメータの引渡しが行なわれる。accept 文の do …… endの間の処理が終わったとき，結果がcallを行なったPmに返され，両者は再び並行処理を始める。同じポートに対し多重にcallが起こったときには，そのポートに待ち行列ができる。call が起きている複数のポートに対し，非決定的処理をするためにガーディッドコマンド流の select 文を用いる。

リングバッファの機能をするモジュール（〔Honeywell 79a〕〔Horeywell 79b〕では task という）は図9-16のように記述される。

図9－14と図9－15を比較すれば明らかなように，構造化交信では，send-receive の順序関係と，ポートの対応関係が正しいことが確実に保障されている。accept 文と select 文を用いると，順路式のサブセットが記述できる。

しかし，交信を構造化することによって本来対称であるはずの状況も非対

称に記述せざるを得ない。処理が必要以上に同期化されてしまう。

```
task ringbuffer is
   entry send ( e : in message ) ;
   entry receive( v : out message ) ;
end ;
task body ringbuffer is
   buffsize : constant integer := 10 ;
   buffer : array ( 0.. buffsize－1 ) of message ;
   inx, outx : integer range 0.. buffsize－1  := 0
   n : integer range 0.. buffsize  := 0
begin loop
      select when n<buffsize =>
                  accept send ( e : in message )
                     do buffer ( inx ) := e end ;
                  inx := ( inx+1 ) mod buffsize ;
                  n := n+1 ;
      or
              when  n > 0  =>
                  accept receive ( v : out message )
                     do v=buffer ( outx ) end ;
                  outx := ( outx +1 ) mod buffsize ;
                  n := n － 1 ;
      end select
   end loop
end ringbuffer
```

図9－16　　構造化交信（リングバッファ）

9.5 言語の設計要件

並行プログラミングが扱わねばならない問題は多岐にわたっており，その多くは今後の研究に待たねばならない。ここではそのうちで最も基本的な同期と通信の機構を中心に，言語の設計の際考慮すべき問題点について述べる。言語の一般的な設計要件に関しては例えば〔wegner 79〕が詳しい。

9.5.1 モジュラリティ modularity

ソフトウエアが理解しやすく保守しやすくなければならないのならモジュラリティは不可欠である〔Bloom 79〕。並行プログラミングに関しては，特にリソースを共有するためにモジュラリティが重要である。リソースは抽象データ・タイプのオブジェクトである。リソース，同期，必要なデータがカプセル化encapsulationされている。提供された操作によってのみリソースにアクセスできる。そのさい同期は保証されている。

モジュール化することによってプログラムの複雑さが減少する。分散システムにおいては，モジュール化をこころがければ自然にプログラムが分解できる。できたモジュールを合理的にプロセッサに対応づけることもできる。

9.5.2 通　信

分散システムのプログラミングを簡単にするためには，同一プロセッサ上のモジュール同志と異なるプロセッサ上のモジュール同志が同じ形式のインタフェイスを持つことが必要である。

共有記憶，メッセージ，手続コールなどの方法があるが，どれが望ましいのかは今後の研究に待たねばならない。手続コールではモジュール間の関係がマスタノスレイブに限られ，パイプラインの関係が記述できない。しかし，直接メッセージを扱う必要がなくなるのでプログラマレベルのセマンティックスが簡単になる。〔Peterson 79〕システムレベルではメッセージ，アプリケーションレベルでは手続コールが望ましいのであろうか。

9.5.3 表現能力と使い易さ

同期と通信の機構としては表現能力がありしかも使い易いものが望ましい。表現能力がないと，OSに頼ったり，非効率的な技法を使う羽目になる。同期の条件が少し変わっただけでプログラミングし直さなければならないのでは使いにくい。〔Bloom 79〕は表現能力があって使い易い機構を選択するための目安を与えている。

同期に使う条件は次の6種類に分類できる。

(1) 要求された操作の種類

(2) 要求された時刻

(3) 要求のパラメータ

(4) リソースの同期状態

(5) リソースの局所的な状態

(6) 履歴の情報

この6種類の条件が書き易いかどうかは次の問題を記述してみればわかる。

(1) リング・バッファ

(2) 優先度がリーダにあるリーダ・ライタ問題

(3) ディスク・スケジューリング

(4) FIFO

(5) 1スロット・バッファ

使い易さとは，制約条件と解の組が互いに独立していることである。そのときには，個々の制約条件に対する解を重ね合わせると，複雑な同期問題の解になっている。制約条件に対する2つの解が違っていたり，個々の制約条件に対するインプリメンテーションが解の別の部分として識別できなければ直交性はみたされない。

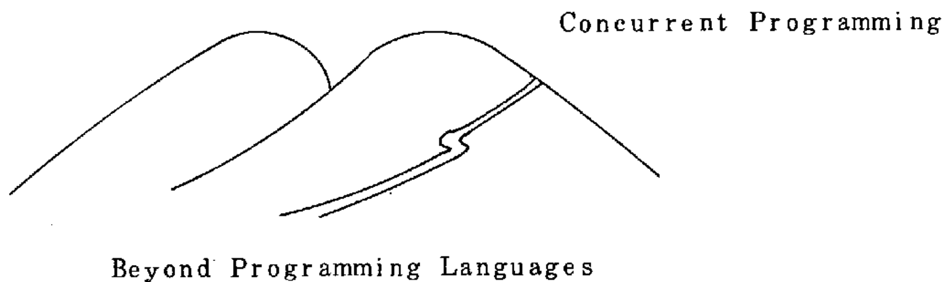
9.5.4 単一言語で十分か

9.3.3で述べたようにデータ・タイプ1つをとっても単一言語で全ての分野

をカバーできないことは明らかである。特に，2.3で述べたようなポリシーとも言える高度の機能を単一言語に持ち込むのには無理がある。Adaが非常に危険な要素を持っている原因は，方言を許さない〔Dod 78b〕万能言語を目指していることにあると思われる。系統的な方言を許す言語ファミリが望ましい。〔Grand 79〕

9.6 あとがき

マイクロ・エレクトロニクスの進歩と分散システムの普及によって，計算機システムは大規模かつ広域化してきている。このような認識のもとに，大規模広域化システムの構築論の確立を目指し，その第一歩として，並行プログラミングの系譜の整理を行なった。



＜参考文献＞

〔Andler 78〕 Andler, S.: Synchronization Primitives and the Verification of Concurrent Programs, Proc of 3rd International Symposium on Orerating Systems Theory and Practice, 67-100 (1978)

〔Andler 79〕 Andler, S.: Predicate Path Expressions: A High-Level Synchronization Mechanism, CMU-CS-79-134 (Aug 1979)

〔Atwood 76〕 Atwood, J.W.: Concurrency in Operating Systems, Computer 9, 10, 18-26(Oct 1976)

〔Ball 79〕 Ball, J.E., G.J.Williams, and J.R.Low: Preliminary ZENO Language Description, SIGPLAN 14, 9, 17-34 (Sep 1979)

〔Balzer 71〕 Balzer, R.M.: Ports - A Method for Dynamic Interprogramme Communication and Job Control, Proc of AFIPS 38, 485-489(1971)

〔Bllom 79〕 Bloom, T.: Evaluating Synchronization Mechanisms, Proc of the 7th Symposium on Operating Systems Principle, 24-32(dec 1979)

〔Brinch Hansen 72〕 Brinch Hansen, P.: A Comparision of Two Synchronizing Concepts, Acta Informatica 1,, 190-199 (1972)

〔Brinch Hansen 73〕 Brinch Hansen, P.: Operating System Principles, prentice Hall(1973)

〔Brinch Hansen 75〕 Brinch Hansen, P.: The Programming Language Concurrent Pascal, IEEE Trans. SE-1, 2,

199-207 (Jun 1975)

[Brinch Hansen 76a] Brinch Hansen, P.: The SOLO Operating System: A Concurrent Pascal Program, Software-practice and Experience 6, 141-149, (1976)

[Brinch Hansen 76b] Brinch Hansen, P.: The SOLO Operating System: Job interface, Software-Practice and Experience 6, 151-164, (1976)

[Brinch Hansen 76c] Brinch Hansen, P.: The SOLO Operating System: Processes, Monitors, and Classes, Software-practice and Experience 6, 165-200 (1976)

[Brinch Hansen 76d] Brinch Hansen, P.: Disk Scheduling at Compile Time, Software-practice and Experience 6, 201-205 (1976)

[Brinch Hansen 77a] Brinch Hansen, P.: Experience with Modular Concurrent programming, IEEE Trans. SE-3, 2, 156-159 (Feb 1977)

[Brinch Hansen 77b] Brinch Hansen, P.: The Architecture of Concurrent Programs, Prentice Hall (1977)

[Brinch Hansen 78a] Brinch Hansen, P.: Distributed Process: A Concurrent Programming Concepts, CACM 21, 11, 934-941 (1978)

[Bryant 78] Bryant, R.E. and Dennis, J.B.: Concurrent Programming, TM-115, LCS MIT (Oct 1978)

[Campbell 74] Campbell, R.H. and Habermann, A.N.: The Specification of Process Synchronization by Path Expressions, Lecture Notes in Computer Science 16, Springer Verlag, 89-102 (1974)

〔Campbell 79a〕 Path Expressions in Pascal, Proc of the 4th International Conference on Software Engineering, 212-219 (Sep 1979)

〔Campbell 79b〕 Practical Applications of Path Pascal in Systems Programming, Proc of ACM 79, 81-87 (Oct 1979)

〔Cook 79〕 Cook, R.P.: *MOD--A Language for Distributed Programming, Proc of the 1st International Conference on Distributed Computing Systems, 233-241 (Oct 1979)

〔Denning 77〕 Denning, P.J.: Forward, Operating Systems Review 11, 5, iii (Oct 1977)

〔Dijkstra 68a〕 Dijkstra, E.W.: Cooperating Sequential Processes, Programming Langauges, F.Genuys (ed.), Academic Press, New York, 43-112 (1968)

〔Dijkstra 68b〕 Dijkstra, E.W.: THE Structure of The 'THE'-Multiprogramming System, CACM 11, 5, 341-346 (May 1968)

〔Dijkstra 71〕 Dijkstra, E.W.: Hierarchical Order of Sequential Processes, Acta Informatica 1, 2, 115-138 (1971)

〔Dijkstra 75〕 Dijkstra, E.W.: Guarded Commands, Nondeterminacy and Formal Derivation of Program, CACM 18, 8, 453-457 (Aug 1975)

〔Dod 78b〕 Department of Defence: DoD PEBBLEMAN Requirements for the Programming Environment for Common High Order Language (Jul 1978)

〔Doi 78〕 Doi, N: On the Path Expression, Joho-Shori 19, 8, 779-787 (Aug 1978) (in Japanese)

〔Grand 79〕 Grand, A.: Issues in tke Design of Concurrent Programming Languages, Proc of ACM 79, 95-101 (Oct 1978)

〔Haddon 77〕 Haddon, B.K.: Nested Monitor Calls, Operating Systems Review 11, 4, 18-23 (1977)

〔Hoare 74〕 Hoare, C.A.R.: Monitors: An Operating System Structuring Concept, CACM 17, 10, 549-557(Oct 1974)

〔Hoare 78〕 Hoare, C.A.R.: Communicating Sequential Processes, CACM 21, 8, 666-677 (Aug 1978)

〔Holt 78〕 Holt, R.C., G.S.Graham, E.D.Lazowska, and M.A.Scott: Structured Concurrent Programming with Operating Systems Applications, Addison-Wesley (1978)

〔Honeywell 79a〕 Honeywell, Inc.and Cii Honeywell Bull: Reference Manual for the Green Programming Language (Mar 1979)

〔Honeywell 79b〕 Honeywell, Inc.and Cii Honeywell Bull: Rationale for the Design of the Green Programming Language (Mar 1979)

〔Hsiao 79〕 Hsiao, D.K., D.S.Ken, and S.E. Madnick: Computer Security, Academic Press (1979)

〔Hunt 79〕 Hunt, J.G.: Messages in Typed Languages, SIGPLAN 14, 1, 27-45 (Jan 1979)

〔Jammel 77〕 Jammel, A.J.and Stiegler, H.G.: Managers versus Monitors, Proc. of IFIP Congress 1977, North Holland, Amsterdam, 827-830 (1977)

[Keedy 79] Keedy, J.L.: On Structuring Operating Systems with Monitors, Operating Systems Review 13, 1, 5-9 (Jan 1979)

[Lagally 78] Lagally, K.: Synchronization in a Layered System, Lecture Notes in Computer Science 60, 252-281 (1978)

[Lauer 78] Lauer, H.C. and R.M. Needham: On the Duality of Operating System Structures, Proc 2nd International Symposium on Operating System Theory and Practice, 371-384 (1978)

[Liskov 79] Liskov, B.: Primitives for Distributed Computing, Proc of the 7th Symposium on Operating Systems Principles, 33-42 (Dec 1979)

[Lister 77a] Lister, A.M.: The Problem of Nested Monitors Calls, Operating Systems Review 11, 3, 5-7 (1977)

[Lister 77b] Lister, A.M. and Sayer, P.J.: Hierarchical Monitors, Software-Practice and Experience 7, 613-623 (1977)

[Lycklama 78] Lycklama, H. and D.L. Bayer: UNIX Time Sharing System: The MERT Operating System, Bell System Technical Journal 57, 6, 2049-2086 (July-Aug 1978)

[Mckeag 76] Mckeag, R.M.: T.H.E. Multiprogramming System, in Studies in Operating Systems, Academic Press (1976)

[Peterson 79] Peterson, J.L.: Notes on a Workshop on Distributed Computing, Operating Systems Review 13, 3, 18-30 (1979)

[Popek 78] Popek, G.J. and C.S. Kline: Issues in Kernel Design, Lecture Notes in Computer Science 60, 209-227 (1978)

[Schutz 79] Schutz, H.A.: On the Design of a Language for Programming Real-Time Concurrent Processes, IEEE Trans. SE-5, 3, 248-255 (1979)

[Uchida 80] Uchida, S. and T. Higuchi: Driven Type Programming, in this report (1980)

[Wegner 79] Wegner, P.: Programming Languages - Concepts and Research Directions, in Reseach Directions in Software Technology, MIT Press, 425-489 (1979)

[Winograd 79] Winograd, T.: Beyond Programming Languages, CACM 22, 7, 391-401 (Jul 1979)

[Wirth 77a] With, N.: Modula: A Language for Modular Multiprogramming, Software-Practice and Experience 7, 1, 3-35 (1977)

[Wirth 77b] Wirth, N.: The Use of Modula, Software-Practice and Experience 7, 1, 37-65 (1977)

[Wirth 77c] Wirth, N.: Design and Implementation of Modula, Software-Practice and Experience 7, 1, 67-84(1977)

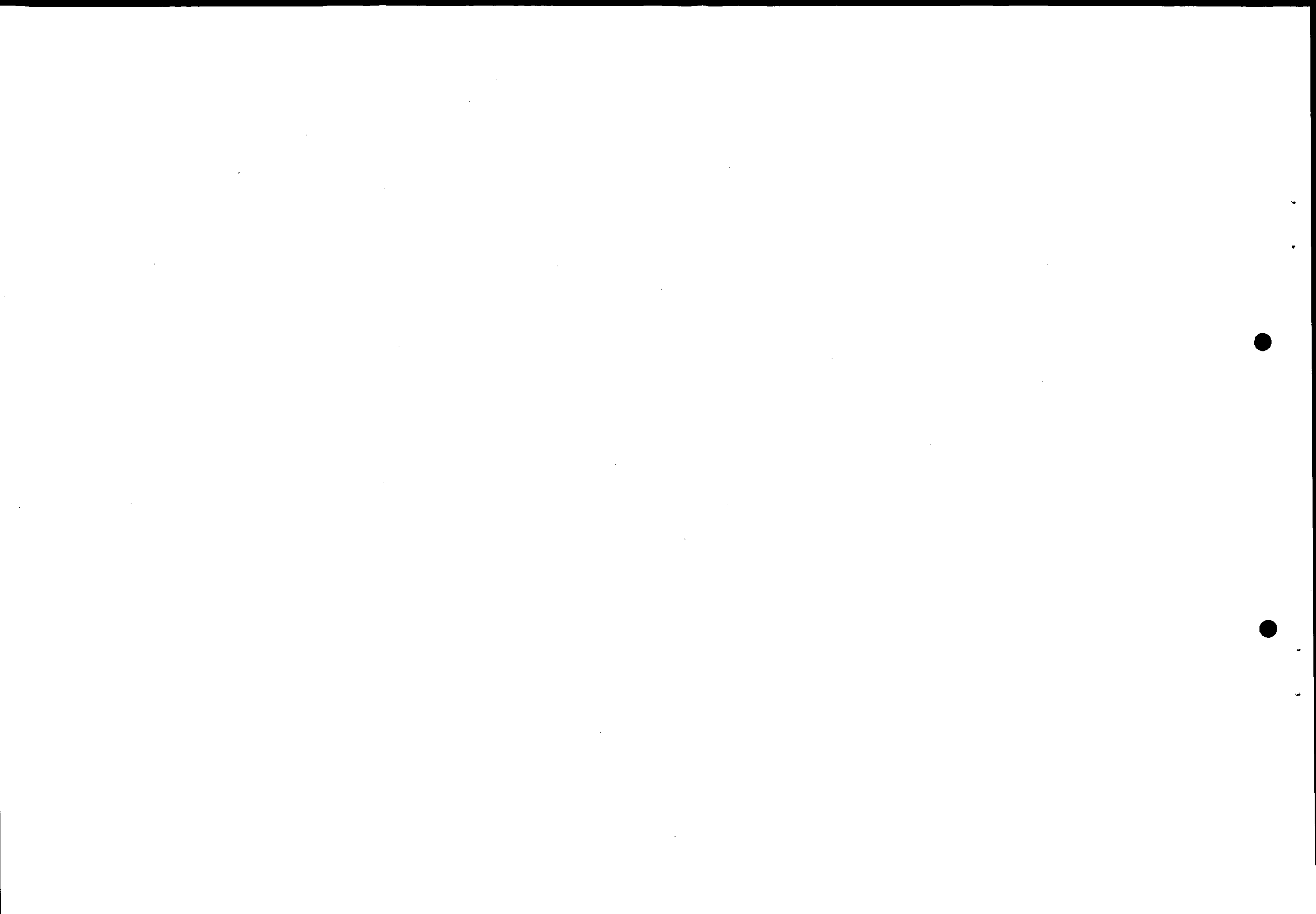
[Wirth 77d] Wirth, N.: Toward a Discipline of Real-Time Programming, CACM 20, 8, 577-583 (1977)

[Yonezawa 79] Yonezawa, A.: Comments on Monitors and Path-Expressions, Joof Information Processing 1, 4, 180-186 (Mar 1979)

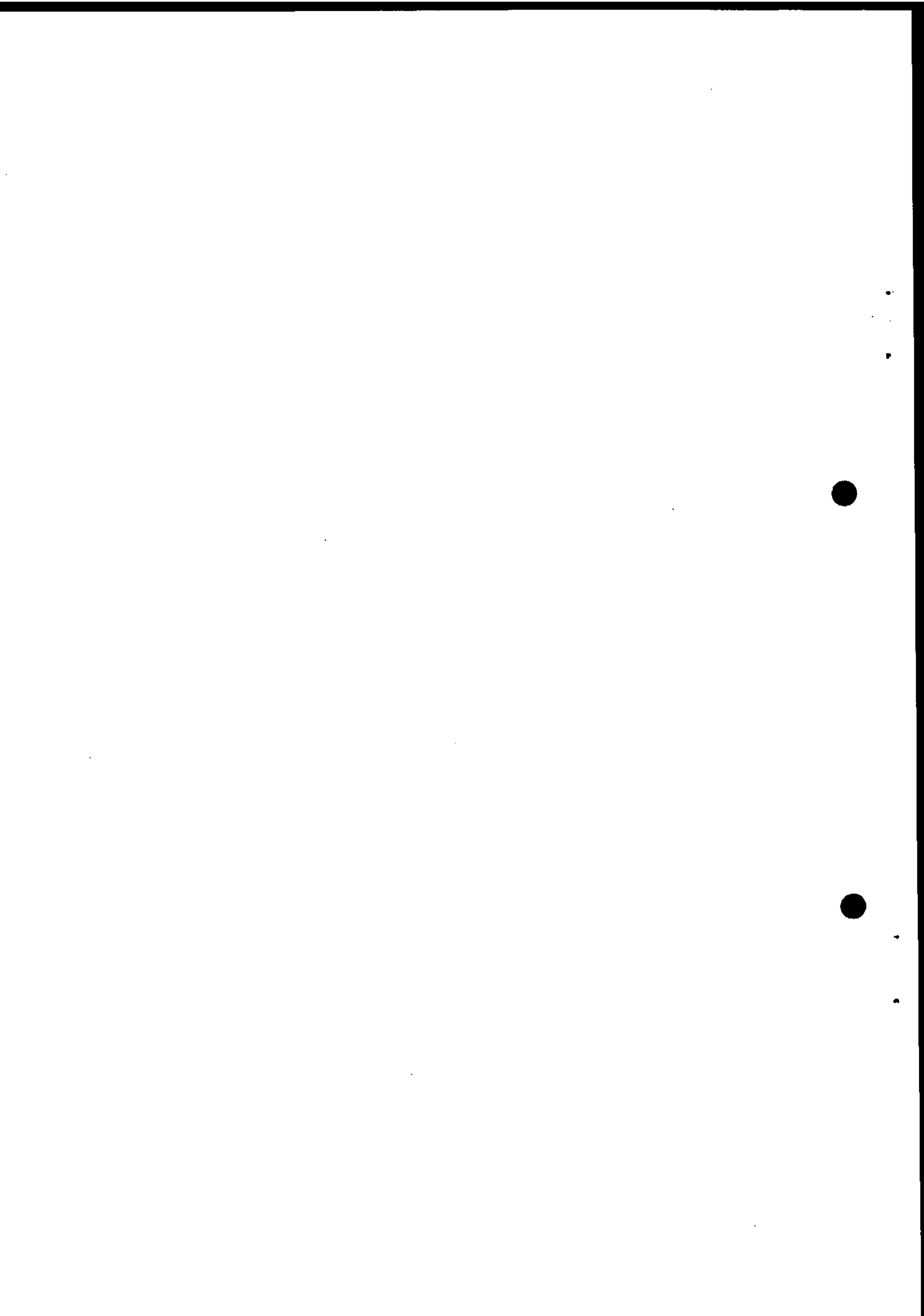
表９－１　代表的な並行プログラム言語

付　録　代表的な並行プログラム言語

	ベース言語	同期・通信の機構	文献	コメント
Concurrent Passcal	Pascal	monitor	〔Brinch Hansen 75〕 〔Brinch Hansen 77b〕	○monitorに制限あり ○モデルOS　SOLOが作られた ○最も著名
Modula	Pascal	monitor	〔wirth 77a~c〕	○一般的なmonitor ○module ○ミニコン（PDP11）リアルタイム用
Iliad	PL／I	conditional critical region	〔schutz 79〕	○素人のリアルタイム用
Masa		port	〔Mitchell 78〕	○ＱＳ作成用
Ada		Structured Communication	〔Honeywell 79a~b〕	○DoDの万能処理用
Path Pascal	Pascal	path expression	〔Campbell 79a~b〕	○システム，リアルタイム用 ○実験用言語
Clu	Sequential Clu	message continuation	〔Liskov 79〕	○分散システム用
＊Mod	Modula	distributed process	〔Cook 79〕	○分散システム用
Zeno	Euclid	port	〔Ball 79〕	○分散システム用 ○連想set ○トランザクション



第10章　対象指向型プログラミング



10. 対象指向型プログラミング

プログラミング言語は，基本となる，行為（ action ）と対象（ object），それらをより複雑な，計算の構造物へと組上げるための，合成の機構からなる。行為に対する合成の機構とは，関数の合成，実行の逐次列をはじめとする各種の制御構造，副プログラム構造などがある。対象に対する合成機構には，配列，リスト，集合等々のデータ構造や，抽象データ構造によるカプセル化の機構，さらにデータ・ベース等がある。

行為中心に計算を表現し，対象を補助的なものとみる見方を行為指向（ action － oriented）といい，対して，対象をより重視する見方を対象指向（ object － oriented）という。数値計算中心のプログラム（ミング，言語）は，行為指向であり，事務計算，データ・ベース処理，高度なデータ構造の処理，さらに記号処理に向うにつれ，対象指向のプログラム（ミング，言語）となってきた。もちろん，行為と対象は，１つのものの２つの側面といえるもので，深い相互関係をもっている。基本行為は，基本対象への基本的な操作に対応し，高度な制御構造は，処理対象となるデータ構造と密接な関連を持つ。

対象指向という見方は，１９７０年代に入り，人工知能研究，ソフトウェア工学において，大きなシステムをどう構成するか，その解決への糸口として重視され，研究された。知識表現における，フレーム（ frame ），自然言語研究におけるスクリプト（ script ），プログラム言語における抽象データ型(abstract data type)，計算モデル，データ・フロー計算機における，アクタ（actor），分散処理システムにおけるモニタ（monitor）やＣＳＰ（Communicating sequential process)等々が提案，研究された。それぞれ，内部の仕組，表現しようとしているものの大小等に相異があるものの，すべてに共通の考え方は，基本となる構成（プログラム，表現）単位をきちんと定め，その単位を集め，組み上げることによって，大きな複雑なシステムを作り易くしようというものである。各々の単本単位の意味は，その内部の表現によって，

十分に表わされて居り，単位間の相互の関係も，色々な方法で明示され，システム全体の構造が把握しやすくなる。

以下に代表的なものについて，簡単に説明する。

10.1 Closure, Actor

λ－計算から見ると，行為と対象は，式，f(x)からの2種類の抽象(abstraction)に対応させることができる。行為は，抽象，λx.f (x)に，対象は，抽象λf.f (x)にである。この対応づけに従って，スタックという対象をλ－計算で表現すると，図10-1のようになる。λ－計算そのものは，非常に単純な構造しかもっていないため，表現されたものは，決して理解し易いものではない。しかし，図のような表現が行なえるのは，自由変数への値の結びつけ（環境）を合せもたせることができる機能と関数とデータが対等に扱える高階の機能による。これらの機能は，対象指向を考える上では，すべてに共通に考慮すべきものである。環境を合せ持ったλ－式をClosure(閉包式）と呼ぶ，この考え方は，直接拡張され，HewittのActor理論のアクタや,Dennis等のデータ・フロー計算機のアクタになる。このアクタは，対象をアクティブなものと考える立場を代表し，ミクロなレベルでの高度な並列処理を実現し，高レベルの汎用プログラミング言語とあいまって，このアクタの集合で，すべての計算を表現しようというのがデータ・フロー・マシンの研究である。

10.2 Class

対象を，その内部状態を表わす局所変数群と，その内部状態を参照したり変更したりするための手続き群の組で表わし，外部からは操作子（手続き名）の列挙によって，対象を定義したようにする機構が，Simula67でClassという考え方として提案された。procedureという行為の抽象化の機構に対し，データ（対象）の抽象化を行なう具体的な手法として評価され，抽象データ型という名称で呼ばれ，新しいプログラム言語が多数提案された。代表例として，Clu

のcluster（図10－2）Euclidのmodule(図 10－3）を挙げる。いずれもスタッフを表わしているが，Popという手続きは図10－1のTopとPopの機能を合せ持ったものである。

$$\begin{aligned}
\mathrm{Create} &= \lambda f.(f(nil))(nil)\\
\mathrm{Top} &= \lambda x.\lambda y.x\\
\mathrm{Pop} &= \lambda x.\lambda y.y\\
\mathrm{Push} &= \lambda x.\lambda y.\lambda z.\lambda f.(f(z))(\lambda f.(f(x))(y))
\end{aligned}$$

（a）

$$(((((\mathrm{Create}\ (\mathrm{Push}))\ (a))\ (\mathrm{Push}))\ (b))\ (\mathrm{Push}))\ (c)$$

$$=$$

$$\lambda f.(f(c))(\lambda f.(f(b))(\lambda f.(f(a))(\lambda f.(f(nil))(nil))))$$

（b）

図10－1　λ－計算によるスタックの記述

```
Stack = cluster [t : type]
  is create, Push, Pop;
  rep = array [ t ];
  create = oper(   ) returns(cvt);
           return(array [t]$ create(1));
  end create
  Push = oper(S:cvt, x:t);
    rep $ extendh(S, x);
    return;
  end Push;
  Pop = oper(S:cvt)returns (t)signals(empty-stack);
        if rep $ size(S)>0
           then return rep $ retracth (S)
           else signal empty-stack;
  end Pop;
end Stack;
```

図10－2　Cluによるスタックの記述

（″情報処理″ Vol20 №1 より）

```
type Stack(StackSize:unsignedInt)=module
  exports(Pop, Push)
  var IntStack:array 1.. StackSize of signedInt
  var StackPtr:0.. StackSize:=0
  procedure Push(X:signedInt)=
    imports(var IntStack, var StackPtr,
            StackSize)
    begin
    procedure Overflow=… end Overflow
    if StackPtr=StackSize then
      Overflow
    else
      StackPtr:=StackPtr+1
      IntStack(StackPtr):=X
    end if
    end Push
  procedure Pop(var X:signedInt)=
    imports(var IntStack, var StackPtr)
    begin
    procedure Underflow=… end Underflow
    if StackPtr=0 then
      Underflow
    else
      X=IntStack(StackPtr)
      StackPtr:=StackPtr-1
    end if
    end Pop
  end Stack
```

図10－3　Euclidによるスタックの記述
（"情報処理" Vol12, No.1より）

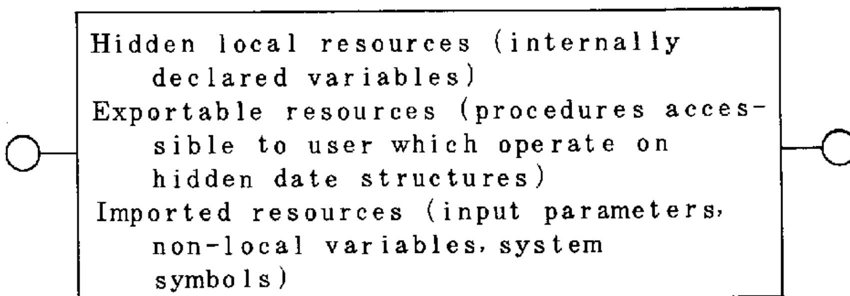


図10－4　抽象データ型カプセル
（"Research Directions in Software Technology"より）

一般に，抽象データ型は，図10－4 のような構造をもつが，具体的な言語仕様は，それぞれ若干の個性を持つ。抽象データ型のタイプ名を何とするかとか，その他の文法上の味つけの好みといったものを除くと，個性を特徴づけるものは，次の2つ位であろう。

(a) カプセル化の度合

特に内部変数への参照を自由化するか否かである。極端なカプセル化は簡単なプログラムの際には，自明な事実の記述を強制することになる。

(b) 抽象データ型の見方

抽象データ型の記述を操作・比較の手続き，例示の仕方等々のそのデータ型に関する記述をすべて集めたものとする見方と，例示されるデータそのものの記述とする見方とがある。前者を代表するものが，CluのCluster で，後者は先のλ－計算による記述をはじめ多くのもの（EuclidのModule)がこれに入る。それぞれ一長一短がある。両者の長所を取り入れた記述系を考案する必要がある。

10.3 抽象データ型の抽象化

代数的仕様記述と称するもので，詳しくは5章を参照されたい。データ型の仕様記述という観点から，文字通り抽象化を一段と進めたもので，代数によって形式的に対象というものをとらえる方法を提案した。一例をスタッフを例にとって，図10－5に示す。

```
type Stack[item]
    declare
        CREATE( )-> stack
        PUSH(stack, item) -> stack
        POP(stack) -> stack
        TOP(stack) -> item
        ISNEWSTACK(stack) -> boolean
    for all s=stack, i=item let
        ISNEWSTACK(CREATE)=true
        ISNEWSTACK(PUSH(s, i))=false
        POP(CREATE)=CREATE
        POP(PUSH(s, i))=s
        TOP(CREATE)=UNDEFINED
        TOP(PUSH(s, i))=i
    and
end Stack
```

図10－5　代数的仕様記述によるスタックの記述

さらに対象（抽象データ型）間の記述法として，階層構造を導き入れ，それぞれ形式化しようという試みがなされている。

対象指向というのは，記述の基本単位についてであり，プログラムにしろ何れにしろ，記述全体が完結するためには，対象間の記述（対象間を記述する対象）等々の構造を考慮しなければならない。このような対象間の構造に立ち入った議論を行なっているのは知識表現の研究分野が最も積極的で，プログラムに関しては，Simulaの初期作業に若干あるものの，本格的な取組みは，仕様記述の研究がはじめてである。

10.4 モニタと順路式

抽象データ型は，手続きの呼び出しに，待合せと同期の機能を付加すると，並列処理の際の記述単位としても利用できる。抽象データ型を並列処理の単位，記述対象である資源に対応させることができるからである。このような機能を持たせた抽象データ型の代表例がHoareのモニタ（monitor）である。待合せの操作を明示するための関数が用意される。さらに，この待ち合せや順序等を抽象化し，明示しようというのが，順路式の考え方で，スタックを例にとり，図10－6に示す。

```
type stack(n=integer)of t=
    array〔1..n〕of t
    const max=n
    var top=integer(0)
    path〔top=0：push, top=max：pop,
        push+pop〕end
    let S=stack, x=t in
      op S.push(x)=S〔top←top+1〕←x
      op S.pop：t=return(S〔1+(top←top−1)〕)
end
```

図10－6　順路式を含むスタックの記述

（"情報処理" Vol19，No.8 より）

10.5 フレームと知識表現

知識を記述対象とする研究分野で，その基本となる記述単位として提案されたのが，Minsky等のフレーム（frame）に代表されるものである。人間のもつ一つ一つの概念が対象に対応づけられる。この考え方に基づいて，具体化された記述言語は，Winograd等のKRL である。一例を図10－7に示す。

IMP	記　　述
Year（年）	Integer
Month（月）	Month-name
Day-number（日にち）	(Integer range(interval min 1 max 31))
Day-of-week（曜日）	Weekday-name
Sequence-number（通し番号）	Integer
ASCII-form（ASCII形）	(Integer length 6 structure(concatenated-repetition* element(integer length 2) number 3))

*繰り返しつなげたもの

（a）　IMP（重要要素）の記述

IMP	記　　述
Year	(Integer structure(concatenation (1) first"19" second (! ASCII structure first)))
Month	(Month-name (position-in-list (2) list"January, February, ………, December" element(! month) number(! ASCII structure second)))
Day-number	(Integer range(interval min 1 max (! month length)))
Day-of-week	(Weekday-name (position-in-list (2) list"Sunday, Monday, ………, Saturday" element (! day-of-week) number (integer range (interval min 1 max 7))) (1-1 correspondence (3) set 1 (! day-of-week position-in-list (2) number) set 2 (quotient-mod-7 (4) dividend (5) (! sequence-number))))
Sequence-number	Integer
ASCII-form	(Integer length 6 structure (concatenated repetition (6) element (integer length 2) number 3)))

(1)　つなげたもの　(2)　リスト内の位置　(3)　1対1対応　(4)　7を法とした場合の商　(5)　被除数

(6)　繰り返しつなげたもの

（b）　IMP間の記述

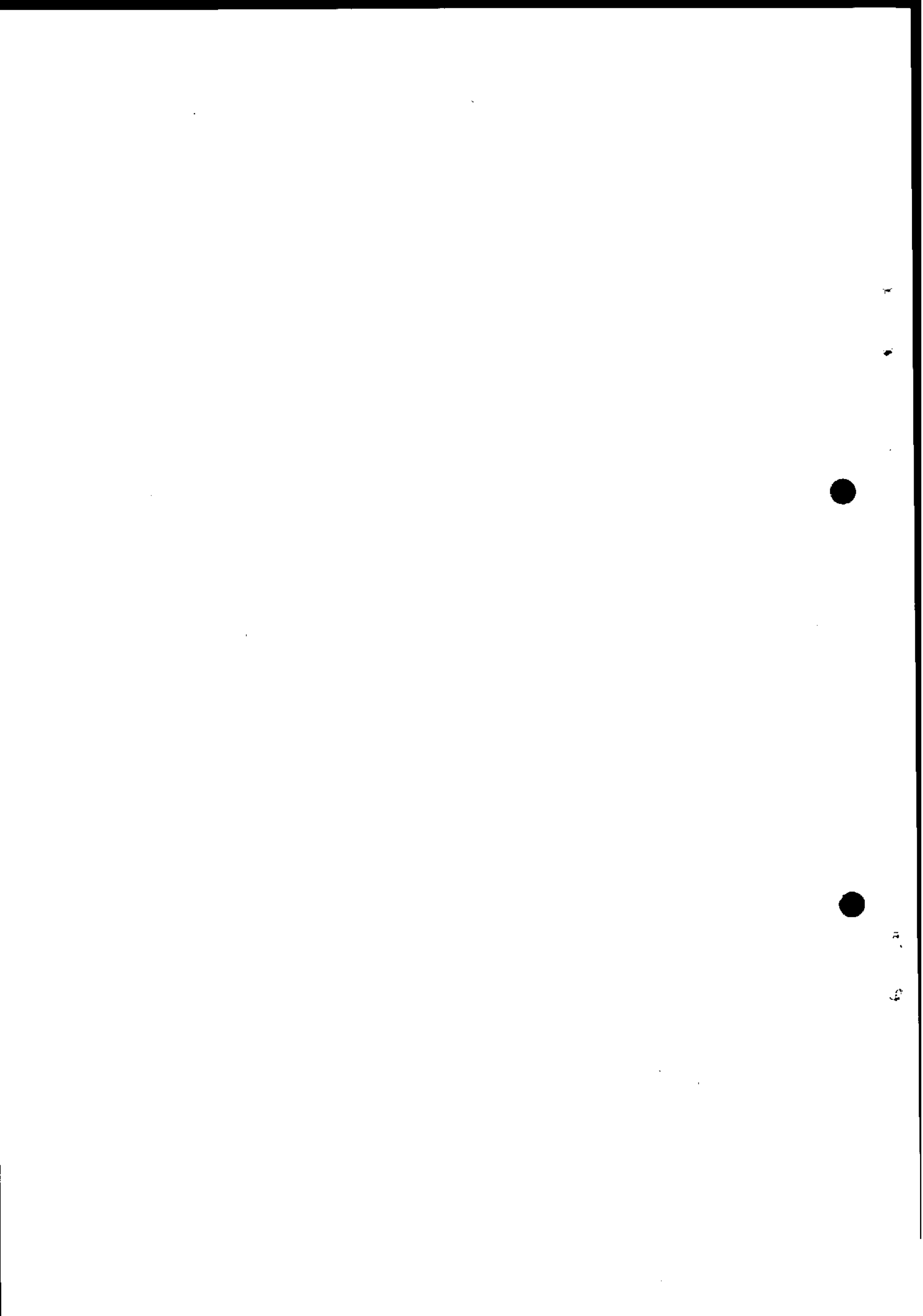
（次頁へつづく）

IMP	手　続　き
Year	
Month	
Day-number	WHEN-FILLED (CHECK-RELATION day-number, month)
Day-of-week	TO-FILL (APPLY calendar-lookup TO year, month, day-number) (APPLY anchor-date-method TO year, month, day-number)
Sequence-number	
ASCII-form	WHEN-FILLED (FILL year, month, day-number)

（c）手続き付加による記述

図10－7　フレームによる日（ day ）の記述
（〃人工知能の基礎〃より）

抽象データ型が，対象の能動的な機能面に関心の重点を置いているが，フレームは，対象の静的な属性面に関心の重点を置いている。もちろん，両者は，融合されるべきものである。ＫＲＬは，対象間の記述（概念の階層関係）等の豊富な構造を取り出したものの，それらをきちんとした計算（推論）機構の上に組み上げるには，成功したとはいいがたい。知識表現，プログラム言語，両者を統合した大きな枠組の中で再検討されるべきである。



第11章　新計算機の骨組

11. 新計算機の骨組

将来の計算機システムを，高度な知識情報処理システムとし，そのハードウエア部分として新計算機の骨組を述べる。

高度な知識情報処理システムには，高機能データ・ベース（知識ベース）と強力な推論マシンが，核となるハードウエアである。

データ・ベースの操作が，計算記述のより大きな部分をになうことになる。大容量データ・ベースの検索もさることながら，動的な小さなデータ・ベースを作り操作するということが，高次の推論には必須になる。そこで基幹をなす，アーキテクチャ技術は，現在，喧伝されているものを用いればデータ・フロー・マシンとデータ・ベース・マシンである。データ・フロー・マシンは，Dennis らの作業を整理し，より一段と発展させたもの，注) データ・ベース・マシンは，関係データ・ベースにおける作業を土台にし，より一段と発展させたものというところが直感的なイメージである。高機能の推論機能を高速に実行するのが，データ・フロー・マシンの役割であり，大容量のデータを蓄え，検索・管理するのが，データ・ベース・マシンの役割である。

注) Dennis 等の現状を全面的に認めるという意味ではない。Hewitt の Actor理論の方がよりすぐれた性質を持っているし，Dennis 等も，その欠点を改善しつつある。ここでは，データ・フロー・マシンを普通名詞として用いる。

具体的には，知識情報処理システムにおいては，知識ベースと推論マシンは，融合したものとして存在し，不可分である。むしろ，一つのものを，データの側面から見た場合が知識ベースで，プログラムの側面から見たのが推論マシンである。ここでは，とり合えず2つのものが存在するとして，それぞれに期待される機能と，その機能が2つのアーキテクチャ技術にどのように関連しているかを以下に列挙する。

(I) 推論マシン⟷(データ・フロー・マシン

データ・ベース・マシン)

(a) 高度な記号処理機能

リスト処理を最も低次のものとして，各種の高度な記号処理をしなければならない。記号としては，集合，バグ，対，数式，論理式等々が想定されるが，いずれも，リスト処理技術をベースにして実現される。リスト処理は，Lisp の上で，副作用なしに，高度なデータ構造の表現と操作を初めて，可能にし，実用化した。したがって，記号処理はデータ・フロー・マシン上で高能率に処理しうる。またリスト処理における記憶領域の動的な配分とゴミ集めの技術は，データ・フロー・マシンの実現のための基本的技術でもある。

(b) パターン照合，連想機能

論理式におけるUnification，引数受渡しの一般化としてのパターン照合，集合等における，等号や包含関係の判定をはじめとする各種記号データ構造の合成，分解等々。低次のものから高次のものまで，各種のパターン照合や連想機能が多用される。この機能は，データ・ベース・マシンの基本機能であるとともに，データ・フロー・マシンの基本実現技術の一つともなっている。また，複雑な記号データ構造の照合等の操作に対しては，専用のデータ・フロー演算器が用意されることとなろう。

(c) 非決定的演算，大データ演算処理→並列処理

計算は，″この内のどれか″という非決定的な表現が多用される。適切なものを選択していく機能が推論機能ともいえる。もちろん低次のものから高次のものまで用意されるであろうが，利用者は，この推論機能を前提としてプログラムを書くようになる。また，多数のデータ要素を一度に，変換，写像するような，大データに対する演算子も多く用いられるようになる。現在の商用計算機（フォン・ノイマン型）に対しては，非決定的演算は通常は，後戻り制御によって，大データ演算は，くり返し制御によっ

て実現されている。しかし，いずれも，単純なものを除いては，非常に効率の悪いものになっている。一方，微細な並列処理を効率良く行なうデータ・フロー・マシンに対しては，これらの演算子を並列制御によって実現することができる。ある意味では，これらの演算は，データ・フロー・マシンの出現によって，はじめて，効率の良い有用な演算機能となりうるということもできる。勿論すべての非決定的処理がすべて，並列処理によるという意味ではない。データ・ベースのTruth Maintenance 機能における，ルール群の適用の場合などの非決定的処理には，やはり後戻り制御が用いられるであろう。この場合の後戻り制御は，実現技術というより，推論機能としての選択の結果とみる方がふさわしい。

(d) 対象指向(object oriented)

プログラムや知識表現におけるモジュラリティを高め，構造化を進める方法として，対象指向の記述形式が色々な方向から提案されてきた。プログラム言語における抽象データ型，仕様記述における抽象データ構造，並列処理記述におけるモニタと順路式，知識表現におけるフレームの考え方等々である。この必要なものを一ケ所にまとめ，他とのインターフェースを明確にする記述法は，データ・フロー・マシンにおける，交信の局所性や他との交信法の明確化を保証し，データ・ベース・マシンにおいては，参照や検索の局所性，他との関係の明確化の保証を与えてくれる有用な考え方である。また，副作用を考慮した対象指向の記述によって，データ・フロー・マシンに，基本的な計算機構をくずさずに，経過依存性を導入することができる。

(II) 知識ベース←(データ・ベース・マシン
データ・フロー・マシン)

(a) データのプログラム化，プログラムのデータ化

データとプログラムの双対性は，受け継ぐべき重要な性質である。この考え方は，データへの参照とプログラムの呼び出しの同一視に進む。Micro

-planner に代表される人工知能向言語における パターンによる呼び出し機構としてまず具体化され，さらにPROLOG に代表される論理プログラミング言語において，一段と整理された。すなわち，プログラム自身も述語論理式として，データと同一の形式にまとめられる。これは，データのプログラム（手続き）化であり，プログラム（手続）のデータ（宣言）化である。この機能は，知識ベースの最も基本となる土台である。知識ベースには事実としてのデータに加え，その事実を支配する一般的な記述（プログラム，ルール，定理）も蓄えられ，推論マシンの処理を受けることにより，高度な演繹を行なうことができる。プログラムのデータ化は，プログラムに関する知識の表現と処理を可能にし，より高階な知識ベースを作ることができる。データのプログラム化は，データをミクロな意味での対象指向としてとらえるもので，データ・ベースへの参照をデータ・フロー概念に統一する重要な考え方である。

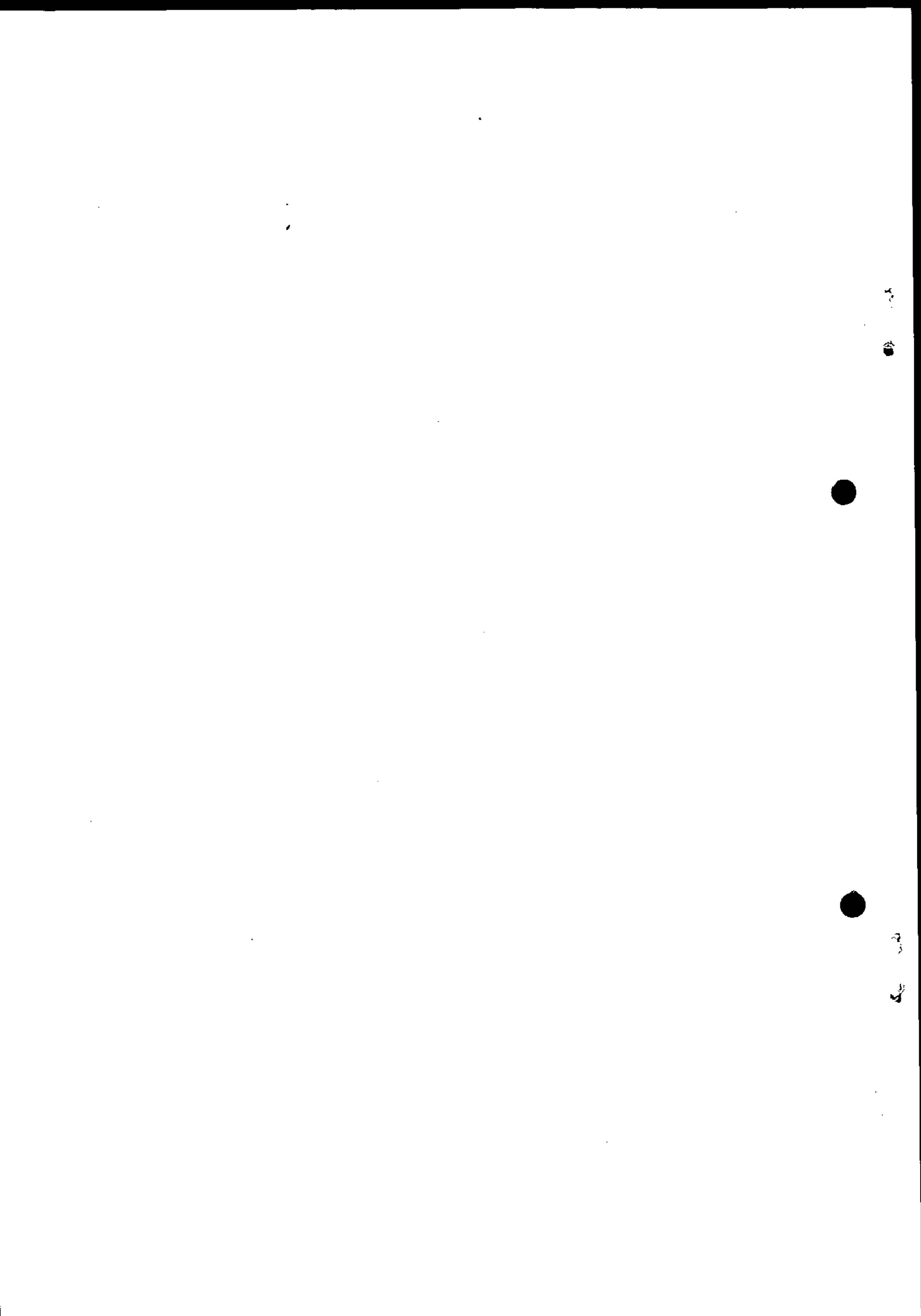
(b) 高度な連想機能

大容量のデータ・ベースの検索のためには，高度なハッシング技術を駆使することになる，また並列検索の機構も必要となる。これらはすべて特殊目的のデータ・フロー・マシンとして設計される。

(c) 高度な代数演算，構造化

関係代数的な演算には，推論マシンの大データ演算と同等の機能が必要になる。また，大規模になるにつれ，蓄積されたデータ群の構造化が要求されるようになる。重要な関係（ Is-a, Part-of 等)を軸にした構造，フレーム的（対象指向的）な考え方の導入等々である。これにより，検索範囲の意味のある限定や，並列検索を高度化することができる。このためには，推論マシンの全機能を駆使することになるし，さらにこの考え方をすすめれば，データ・ベース・マシンのデータ・フロー化に結びつく。例えば，意味ネットワークのハードウエア化は，その兆候の一つである。

第12章　新プログラム言語の構造



12. 新プログラム言語の構造

PL/I，ALGOL68という姿で一応の完結を見た汎用プログラム言語が，次世代（第5世代）に向けて大きく変貌し，新たな姿で提案されうる諸条件がととのってきた。この10年間の人工知能研究，プログラム基礎論における原理的な段階での研究成果の蓄積，Lispから人工知能向言語，関数型プログラミングから論理プログラミング等々での具体的な提案や使用経験の蓄積が変貌の土台を形成した。

重要なことは，新しい情報処理システムを考える際には，まず設定されなければならないのは新しいプログラム言語である。その理由は次の3点による。

(1) そのシステムが対象とする各種の応用のすべてに共通の機能をまとめあげたものがプログラム言語である。

(2) 汎用計算機は程度の差こそあれ，高級言語マシンとして設定されてきた。今后，この傾向は一段と強まる。新しい高度な言語の提案と新しい計算機の構想は一体をなすものである。

(3) ソフトウエア生産性向上のための技術開発の掛け声も，いささか新鮮味を失ってきた。もう一段飛躍するためには，既存言語から離れ，新しい言語の上で議論を展開しなおす必要がある。

さて，それでは第5世代の汎用プログラム言語の構造がどのようなものとなるか，その特徴を列挙するかたちで以下に述べる。

(a) 核言語

核となる汎用プログラム言語は，ほぼ一つのきれいな体系にまとまるであろう。しかし，一般の利用者に提供される段階では，利用者の能力，用途，適用分野によって様々な，それぞれに最適の姿をとる。図形，表，自然言語etc.とバラエティに富んだものとなる。どのような姿をとろうとも，核言語との対応づけにより，意味が厳密に定まり，相互の変換が自由になる。

(b) 仕様，証明，意味

今までのプログラム言語が，コンパイラの作成法，文法解析アルゴリズム，プログラムの最適化に重点が置かれたが，これからは，仕様記述，正当性の証明，意味論等に重点がうつる。そのような観点から，プログラムの最適化や変換が議論されることになる。

(c) 記号処理

従来の数値計算中心の言語から，記号処理中心の言語に移行する。高度な知識情報を扱うための基本的な記述系を与える。数値計算は，数式処理に少しづつ置き換っていく。もちろん，数値計算そのものが無くなるという意味ではない。大規模な数値計算への要求はむしろ増していくであろうが，全体に占める比率は大きく減少していく。また，この記号処理機能によって，言語自身の仕様，証明，意味等の機械的な処理を自分自身の上で一様に行なうことができる。

(d) データ・ベース

事務計算は，単なるファイル処理から，知的なデータ・ベースへの操作と記号処理に置きかわっていく。すべての計算において，データ・ベースの占める役割は大きくなる。プログラムやドキュメント・ライブラリも知識ベース化される。

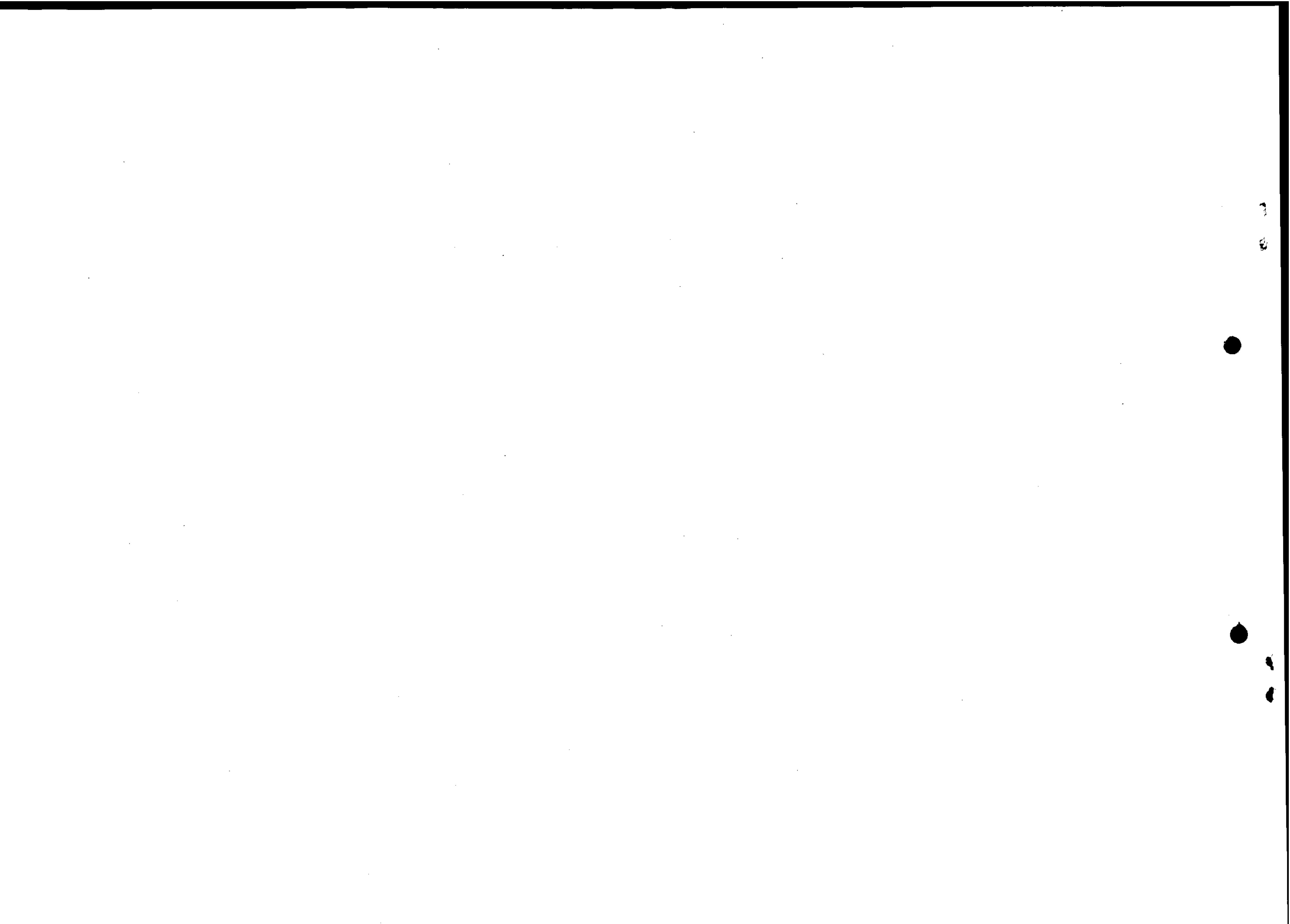
(e) プログラミング・システム

コンパイラ，インタプリタ，デバッグ用支援システム，検証システム，データ・ベース等が一つのまとまったプログラミング・システムの形をとる。各サブ・システムへのアクセス機能が一様な形で言語仕様中に含まれる。

(f) 会話型

インタプリタのモードでは，タイプ宣言無しの自由度の高い会話型言語である。タイプ宣言は，適宜つけることができ，高速化，最適化，誤り・検出のためにシステムが用いる。

第五世代 計算機の体系(一断面)						
数 学		プログラム基礎論		プログラム言語		計 算 機
∘記号論理学 述語論理 様相論理 モデルの理論 λ一計算 コンビナトリ論理 ∘集合論 圏理論 束 論 代数系	∘新しい理論構築の材料を提供する。特に記号論理系の内部構造が明らかにされる。 ← → ∘理論的整理の枠組を与える。	∘意味論 ①記述の階層 公理論的 代数的 示指的 操作的 ②記述の対象 制御構造 データ構造 データ・ベース 並列(非決定的)処理 ∘(プログラム)論理系 アルゴリズム論理 内包論理 ∘計算機構論 関係(論理)型 関数型 並行型 オブジェクト志向型 ∘構成(作成)論 (問題解決理論)	両者がほぼ融合するのではないか。 ←→	∘高度なデータ構造 リスト，対，集合，論理式，数式 etc. ∘いくつかの一般的な推論機構を前提とした制御構造 非手続き的，宣言的，論理的……なプログラム記述 ∘データ構造と制御構造の融合 高次なオブジェクト志向 ∘高機能データ・ベースとの融合 関係モデル・知識ベース ∘一つの論理系として整合性のある体系となっている。 ∘表現要素の意味が厳密に定められ，きちんとした構成論の上に立つ。 ∘文法は，自然言語的なもの，図的なものを含め，多種のものが用意され，相互の変換，対応も厳密に定義されている。 ∘自分自身も含め，各種システムを記述するための，高度な記号処理機能を持つ。 ∘高度なプログラミング・システムとしてまとめあげられている。 (Verifier, Symbolic Execution Debugger, Semantic Editor, Knowledge Base)	∘基本計算機構として並行(駆動型)計算機構を採用することを可能にする。 ∘高速・大容量の記号処理機能を提供する。 ← → ∘高級言語計算機としての具体的な枠組を与える。 ∘ハードウエア記述・システム記述，シミュレーション，ドキュメンテイション等，新しい設計，製作支援システムの体系を与える。	∘プロセッサ 超密結合ポリ・プロセッサ系 (データ・フロー計算機……) ∘システム 多階層ネットワーク系 (分散処理システム……) ∘特性 ①並列計算による高速化 ②高信頼性 ③整構造性(VLSI技術の利用) ④可変構造性 ∘機能 ①高能力な計算機構 ②記号処理機能 ③パターン照合，検索機能 ④高性能データ・ベース



(g) システム作成

言語の高級な機能に制限を加えたようなシステム記述言語が用意され，システム全体が一様な言語系で作くり上げられる。システムの変更，拡張が高級な利用者に完全に開放される。

(h) 透明な計算機構

論理型，関数型を２つの柱として，両者を融合する計算機構も設定され，非常に簡明な原理に基づいたプログラミング言語となる。より宣言的になり，より非決定的な表現が用いられることになる。高度な制御構造も導入されるが，基本制御構造の自然で意味づけの厳密な合成物として定義される。推論機能，連想機能も，この計算機構上に自然な姿で組上げられる。

(i) 並列処理

基本となる計算機構は，逐次計算モデルから並列交信モデルに置き換えられる。言語の意味が，より数学的なものに近いものとなる。副作用は極力排除され，有意な副作用はカプセル化され，その意味づけが明確にされる。高機能の並列処理に対する記述要素も用意されるが基本計算機構と一様に結びついたものとなる。

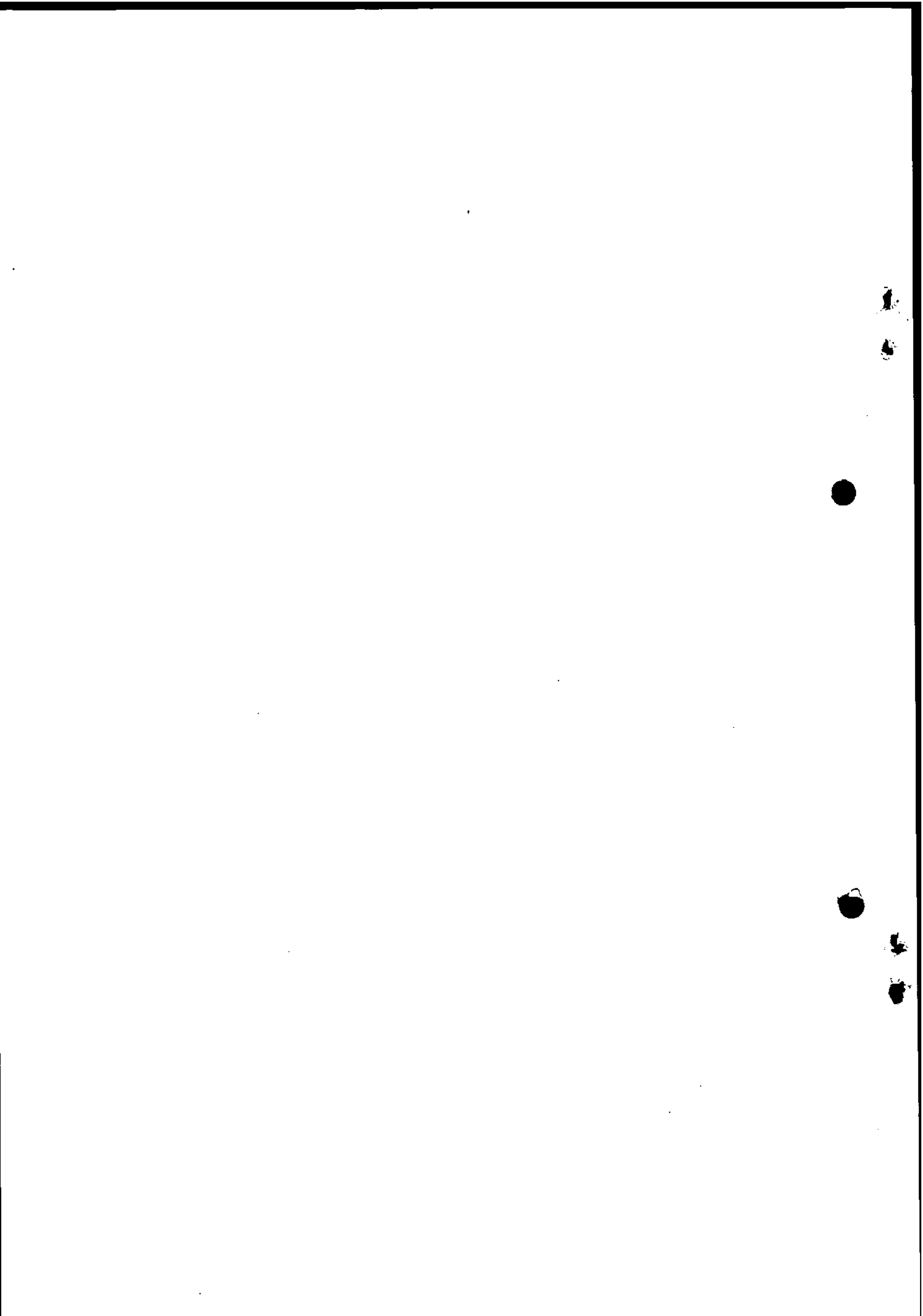
(j) 対象指向

形式的な仕様記述を拡張，発展させた抽象データ型の記述法が，計算表現の上でもデータ・ベースに対しても大きな役割を占めるようになる。また，集合をはじめとする大機能のデータ構造も用意され，データの手続き化，手続きのデータ化が計られる。対象指向と行為指向は，一つに融合されたものとなる。

－－－－－－－－－－－－－－－－－－－－

理論，言語，計算機の結びつきを図１２.１に示す。

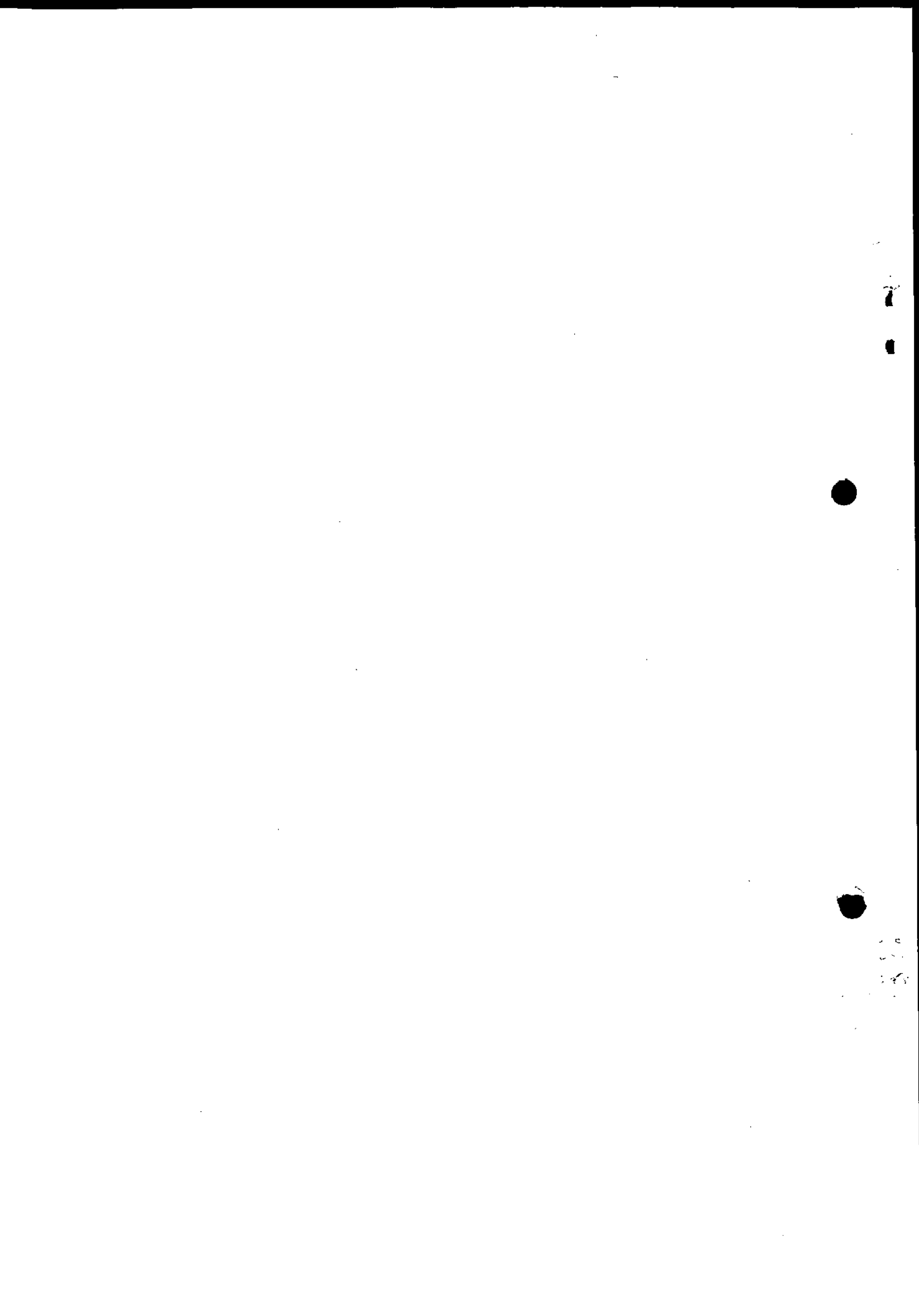
第13章　むすび



13. むすび

筆者達の精力的な取組みにもかかわらず，時間の都合上や，いまだ未消化のため，不十分な点や整合性に欠ける点もある。近い将来，より完成された形で世に問うつもりである。

しかし，現状のままでも，これ程多方面から，広く深く計算の機構を論じたものは無く，これからの計算機研究の大きな糧となるものと自負する。



―― 禁無断転載 ――

昭和55年3月発行

発行所　財団法人　日本情報処理開発協会
東京都港区芝公園3丁目5番8号
機械振興会館内
TEL (434) 8211 (代表)
印刷所　㈱イフ・アドバタイジング
東京都千代田区隼町1番地
TEL (262) 2984 (代表)